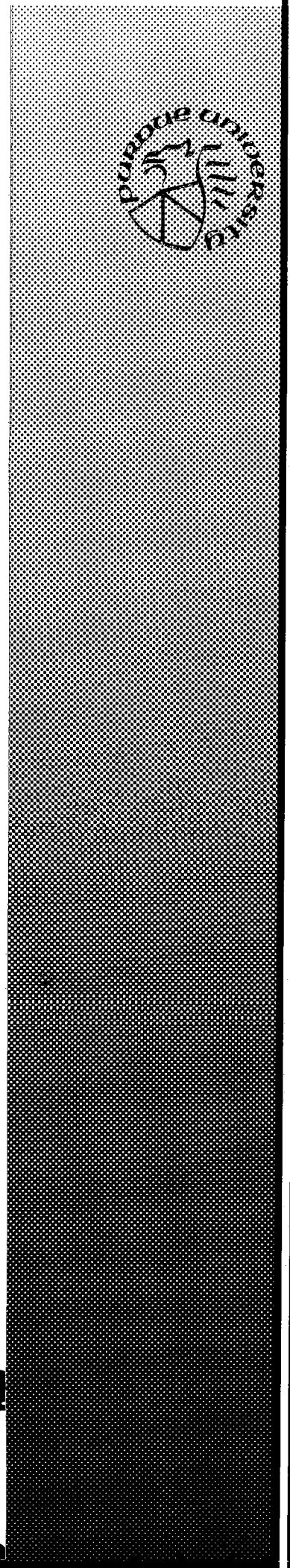
4-1-1988

Emulating a CISC with GaAs Bit-Slice Components

Erin Handgen
Purdue University

Bryan Robbins
Purdue University



Emulating a CISC with GaAs Bit-Slice Components

E. Handgen
B. Robbins

TR-EE 88-11
April 1988

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Emulating a CISC with GaAs Bit-Slice Components

Erin Handgen and Bryan Robbins

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

TR-EE 88-11
April 1988

This research (especially in its final phases) was partially supported by NCR Corporation, World's Headquarters, Dayton, Ohio.

Table of Contents
Emulating a CISC with GaAs Bit-Slice Components

1.) Emulating a CISC with GaAs Bit-Slice Components:
Essence of Problem and Solution
Erin Handgen, Bryan Robbins, and Veljko Milutinovic

2.) Appendix A: Microinstruction Fields
Erin Handgen and Bryan Robbins

3.) Appendix B: Simulation Program
Erin Handgen and Bryan Robbins

4.) Appendix C: Microprogram Data
Erin Handgen and Bryan Robbins

Emulating a CISC with GaAs Bit-Slice Components

Currently, complex instruction set computers (CISCs), are used in a wide variety of applications ranging from desktop computers to aircraft guidance systems. The amount of software available for these CISCs is rather large, and exists in both high-level language and assembly language forms. However, some system designers may need CISCs that are more environmentally tolerant and/or faster than current silicon processors, and may be looking towards GaAs as a possible solution. Aerospace engineers and defense engineers are among this elite group. Some areas where the advantages of GaAs are needed over current Si technology include stabilization and control, telemetry and command, antenna control, and failure detection.[1]

In this work, an attempt is made to draw some general conclusions on how GaAs bit-slice components can be used in a microprogrammed machine to achieve a speed increase over silicon. In particular, the MC68020 has been emulated using state-of-the-art 2900 series bit-slice components from Vitesse Semiconductor Corporation. However, before presenting the solution, we will first define the problem.

It is already well known that GaAs has several advantages over silicon. Besides being faster than ECL, the fastest Si technology, GaAs is more environmentally sound. GaAs can operate in radiation levels above 100 million rads, and its operating temperature range* extends from about -200 °C to about 200 °C.[2] Also, GaAs can interface directly with optical fibers, making it ideal

* Some of the parts used in this research operate in the same temperature range as Si.

for high-speed communications.[3]

Unfortunately, GaAs also has several drawbacks compared to Si. Besides being much more expensive, it has a higher density of dislocations, resulting in smaller VLSI area, smaller transistor count, and above all, a worse yield. Also, GaAs is more costly due to the scarcity of gallium and the fact that GaAs is a compound. Further disadvantages of GaAs include a lower noise margin, brittleness, and a limited availability of appropriate high-speed testing equipment.[3] Despite the apparent inadequacies of GaAs, it is still used to improve system performance in selected applications.

Because of the above mentioned limitations, it is not possible to implement an entire CISC (like the MC68020) on a single GaAs chip. Only multiple chip implementations are feasible. There are three basic approaches to the design of a GaAs based CISC:

1.) Divide the processor into bit slices, and fabricate each slice as a chip

2.) Functionally divide the processor and fabricate each function as a separate chip

3.) Emulate the processor using a RISC type architecture

The RISC-oriented approach was investigated by Kevin McNeley and Veljko Milutinovic at Purdue University.[4] This approach works best for programs written in high level languages, but was not as efficient as initially anticipated for machine language CISC code.[4]

Our research centers on a combination of the bit-slice and functional approaches. Such an approach can emulate the machine language of the CISC without the need for an optimizing compiler or translator which is needed for the RISC approach.

As stated before, our task was to emulate the MC68020 using GaAs bit-slice parts. First, a brief overview of the target machine (MC68020) is given, including the features which were not emulated in our design.** Then, a very detailed description of the host machine is presented, including a timing analysis of the critical path. The core results are given in tables which compare the number of cycles needed to execute target level instructions on the MC68020 and on the GaAs host (through emulation). Finally, suggested improvements and alternate design philosophies are discussed.

The Target Machine:

This section describes briefly the key features of the MC68020. These features include:

- Eight 32-bit general purpose data registers
- Eight 32-bit general purpose address registers
- 32-bit program counter
- 18 addressing modes
- Operations on seven data types

The MC68020 is organized into basic units which operate in a highly independent fashion, maximizing efficiency and performance.[5] A block diagram of the MC68020 is shown in Figure 1.

The bus controller loads instructions from the data bus into the decode unit and the instruction cache. The sequencer and control unit provide overall chip control, including the internal busses, registers, and functions of the execu-

** We concentrated only on features that were relevent for the efficiency analysis.

tion unit. Synchronization of all units is also controlled by the sequencer and control unit.[5]

The execution unit carries out all logical and arithmetic operations involved in the instruction execution. This is also accomplished by means of special hardware, such as a barrel-shifter that allows shifts to be performed in a fixed time, independent of the number of bits to be shifted.[6] Since the basic blocks operate independently, operations can occur simultaneously, allowing some instructions to execute in "zero time". For example, while the execution unit is finishing a computation, the sequencer can be decoding the next instruction.

The basic data types supported by the MC68020 are byte, word, long-word, and quad-word integers, packed and unpacked BCD numbers, bits, and bit fields of 1 to 32 bits. The registers which are available to the user are shown in Figure 2.

Registers D0-D7 are used as data registers for all the basic data types. Registers A0-A6 can be used in some word and long word data computations, but are used primarily to hold pointers to memory such as operand addresses. Register A7 is the user stack pointer, which is used implicitly in all instructions involving user stack operations. All of the data and address registers can be used as index registers in addressing modes which require an index.

The condition code register is 16 bits, but only the lower byte is available to the user. The condition codes are handled automatically by the CPU during logical and arithmetic operations, and each can be tested by the user. The condition code bits are: extend (X), negative (N), zero (Z), overflow (V), and carry (C). The extend flag is used in extended precision arithmetic.

Each instruction of the MC68020 is one word long and may be followed by up to ten extension words. The extension words may be used to specify

displacements and registers to be used in generating the effective address of an operand.

The MC68020 supports a wide variety of addressing modes. These modes were created to help high level language compilers access arrays, records, and linked list data structures.[6] The simplest modes are the data and address register direct modes. These specify operands which are stored directly in the registers. Similarly, immediate data is found in the extension word(s) of the instruction.

There are four address register indirect modes which use an address register as a pointer to where the operand is located in memory. Two of the modes automatically increment or decrement the address register so it points to the next operand. This is useful for vector or stack operations. Another mode adds the contents of the address register to a base displacement to form the address of the operand.

Most of the remaining addressing modes use a base displacement, index register, and outer displacement along with either an address register or the program counter to form the address of the operand in memory. Any or all of these elements can be suppressed to create a variety of related addressing modes.

The MC68020 instruction set contains over 100 instructions which are designed to support high level language compilers, system control, and multiprocessor synchronization. The instruction set can be divided into seven groups: data transfer, arithmetic operations, logical and shift operations, bit and bit field operations, array processing, program flow control, and CPU control.[6]

The data transfer instructions include several forms of move instructions as well as instructions to calculate the effective address of an operand and store it

in a register or the user stack.

The arithmetic operations are add, subtract, compare, negate, clear, sign extend, multiply, and divide. Also, there is a pack, unpack, negate, add, and subtract for BCD operations.

The logical instructions are AND, OR, XOR, and NOT. There is also a test function which compares an operand to zero and sets the condition codes. The shift and rotate instructions are arithmetic shift right and left, logical shift right and left, rotate right and left, and rotate right and left with the extend bit. The swap instruction exchanges the upper and lower word of a data register.

The bit and bit field instructions allow a range of bits to be set, cleared, tested, and complemented.

The array instructions check an operand to see if it lies between two bounds. If the limits are exceeded the condition codes are set and a trap occurs.

The program flow control instructions include jumps, subroutine calls, subroutine return, conditional branches, and sophisticated procedure calls that facilitate high level language parameter passing.

The CPU control instructions include privileged instructions, condition code register manipulation instructions, exception processing instructions, and multiprocessor support instructions.[5] The multiprocessor support instructions are indivisible instructions, such as test and set, that are performed on shared variables in a multiprocessor system.

As indicated and explained before, some features have not been emulated in this design. These features include: supervisor modes, coprocessor support, cache memory, bus arbitration, bus exception, and interrupts. Also, the three

cycle asynchronous memory access of the MC68020 was replaced by a simpler single cycle memory access. This assumption was made since GaAs memories have access times seven times faster than the minimum cycle time of the host, as will be shown later.

The Host Machine:

The host machine was designed using GaAs parts from Vitesse Semiconductor Corporation and Gigabit Logic Corporation. In order to understand the descriptions to follow, the reader should be acquainted with the associated data sheets from these vendors. The bit-slice parts are made by Vitesse and include the VS29G01 4-bit microprocessor slice and the VS29G10A microcontroller. The registers, multiplexers, buffers, and basic logic gates are made by Gigabit. Any custom parts that were needed could be constructed using Gigabit's standard cell library, and the foundry facilities supplied by both Gigabit and Vitesse. Before describing the host machine, it is worth describing the bit-slice parts from Vitesse, the VS29G01 and the VS29G10A.

The VS29G01 block diagram is shown in Figure 3. The circuit is a 4-bit microprocessor slice that is functionally equivalent to the Am2901C.[7] The two main components are the 16 word by four bit 2-port RAM and the eight function ALU. The four bit A address and B address fields select the words that will be output to the A-port and B-port of the RAM. The 9-bit instruction field has three, 3-bit sections. The source field, shown in Table 1, controls the data source selector which selects the two ALU source operands R and S. The possible sources are the A-port and B-port of the RAM, direct data input, logic zero, or the Q register. The function field, shown in Table 2, selects one of the eight ALU functions. The ALU also produces the carry, negative, overflow, and zero condition codes of the result. The destination field, shown in Table 3, controls

how the result of the ALU function is written back into the RAM and the Q register. Both the RAM and the Q register have shifters that allow their contents to be shifted one position right or left before being stored.

The VS29G10A microcontroller is a microinstruction address sequencer which controls the sequence of execution of microinstructions stored in the microcontrol store (μCS).[8] It is functionally equivalent to the Am2910A, with the exception of the Output Enable (OE) input which is active high.[7] The block diagram is shown in Figure 4. A four-input multiplexer is used to select either the register/counter, the direct input, the microprogram counter (μPC), or the stack as the source of the next microinstruction address. The register/counter is a 12-bit latch which can be used to control microprogram loops. The direct data input is used for subroutine calls and branching. The μPC is a 12-bit register that can be used in one of two ways. When the carry-in to the incrementer is high, the μPC is incremented. When the carry-in is low, the μPC is unchanged and is passed to the multiplexer. The stack is 9 words deep by 12-bits wide and is used for subroutine and loop support. A built in stack pointer always points to the last word pushed on the stack, allowing looping to be performed without a pop.

Now that the ALU slice and microsequencer have been described, the host machine can be discussed. The host machine can be broken up into three basic sections: the control unit, the ALU section, and miscellaneous support circuitry.

Control Unit:

The control unit is shown in Figure 5. The control unit controls all actions of the machine by reading and decoding a target level instruction, and then generating the proper sequence of control lines for the host machine. Instructions are read from memory and put in the prefetch instruction register (PIR).

The PIR is a 16-bit register which holds the next target level instruction to be executed. The PIR is used to overlap the instruction fetch of the next instruction with the execution of the current instruction. When the next target instruction is ready to execute, the 16-bit IR is loaded with the data in the PIR. Any extension words that are required by the instruction are held in the extension word register (EWR). The effective address register (EAR) is loaded with a 6-bit effective address contained in one of two fields of the IR.

The instruction register PROM (IR PROM) uses the PIR to generate starting addresses of the microprograms corresponding to each target level instruction. The μPC is loaded with the starting address, and the microsequencer then executes the microprogram in the μCS pointed to by the μPC. The μCS sequentially outputs the microinstructions for each target level instruction, and the microinstructions are stored in the pipeline register.

When the microsequencer performs a jump or a subroutine call, the next address can come from one of three sources, which are enabled by a 2:4 decoder. If the microsequencer is jumping to a new microprogram, then the source is the IR PROM. If the microsequencer is calling an effective address subroutine, then the starting address comes from the effective address PROM (EA PROM). Finally, if the microsequencer branches during a microprogram, the next address comes from the pipeline.

The microsequencer has several conditional instructions. The condition code (CC) multiplexer selects a bit in the host machine which is to be tested. This condition code bit is used to determine the outcome of the conditional microsequencer instruction. The sources to the multiplexer (to be explained later) are: EWR, instruction register (IR), lower internal data bus (LIDB), Bcc circuit, and the host status register. A logical one and logical zero may be used to force the condition into a known state. Also, an XOR gate allows the condi-

tion from the CC MUX to be tested as active high or active low.

The OP SIZE register is used to indicate the size of the target level operation to be performed. It can be used to pass information to microsubroutines, such as the effective address subroutines. For example, the address register indirect predecrement addressing mode subroutine uses the information in the OP SIZE register to determine whether it needs to decrement the address register by 1, 2, or 4. It can be loaded from one of four sources: the pipeline, IR bits 7-6, IR bits 10-9, or IR bits 13-12.

ALU section:

The ALU section is shown in Figure 6. There are two internal data busses of the ALU section, the upper internal data bus (UIDB) and the lower internal data bus (LIDB). The two bus architecture separates the outputs and inputs of the two ALUs so the critical timing path does not go through both ALUs. This reduces the cycle time of the host by a factor of about two. Also, some parallelism is achieved, as one ALU can use the UIDB while the other uses the LIDB. As an example, the address ALU can load the MAR while the data ALU receives data from the DBI.

The address ALU, shown in Figure 7, consists of eight VS29G01 4-bit microprocessor slices and a carry look-ahead unit. The registers in the address ALU hold the target level address registers A0 to A7 and the PC. The carry-in is a function of the address ALU instruction bits. Whenever the address ALU instruction is an add, the carry is cleared. When the instruction is a subtract operation, the carry is set so all subtract operations are 2's complement arithmetic. The latched negative (N) flag from the host level status register is inverted and connected to the least significant bit of the Q shifter. This sets the bits of the Q register during a target level divide operation. The most

significant bit of the address ALU Q shifter is connected to the least significant bit of the data ALU Q shifter to allow 64-bit shifting operations.

The data ALU, shown in Figure 8, is very similar to the address ALU. Two of the main differences are the data ALU masking circuit and the condition code (CC) size multiplexer. The masking circuit is used to disable the most significant ALU slices during certain byte and word operations. This is done by passing a no-operation instruction (NOP) to the disabled ALUs. In a similar way, the CC size multiplexer selects the proper condition codes to be passed onto the target and host status registers. As an example, during a target add operation, where the destination is a word sized data register, the upper four ALU slices will be disabled so only the lower word of the destination data register is modified. Also, the CC size multiplexer will select the condition codes generated by the lower four ALU slices. The carry-in to the data ALU section can come from either the pipeline register or from the extend flag of the target status register. The most significant bit of the Q shifter is connected to the least significant bit of the RAM shifter, to support the divide instruction.

There are two status registers. The host status register contains the condition codes of the data ALU. These condition codes are used by the microsequencer to make branching decisions. The target level status register contains the condition codes of the target machine. The target status register multiplexer determines the source of the condition codes. Possible sources for the target condition codes are the data ALU, the address ALU, the UIDB, and the shifter.

Both ALUs have multiplexers that select from eight possible sources to determine the A-port and B-port RAM addresses. These sources come from different fields of the PL, IR, EWR, and the EAR. The PL fields allow the microprogram to select specific RAM addresses such as the PC. The IR and

EWR fields are 3-bit fields in the target level instruction and extension word that point to D0 through D7 and A0 through A7. The EAR is a latch that is used to point to the base data or address register for the effective address calculating microsubroutines.

The VS29G01 ALU is able to efficiently emulate many of the instructions used by the MC68020. However, a few instructions required some extra functional accelerators to improve the efficiency of emulation. These are the feedback latch, shifter, and multiplier. The first two devices could be designed using a custom or standard cell approach. The multiplier may not be as easy to implement. See references 9, 10, and 12 for some work done on 16 x 16 bit, 12 x 12 bit, and 8 x 8 bit multipliers in GaAs. In this research, two methods of multiplication were used. The first assumed that a 16 x 16 bit GaAs multiplier from reference 9 could be used. With a 16 x 16 bit multiplier, a 32 x 32 bit multiply could be calculated by summing together four 16 x 16 bit partial products. A block diagram of the circuit is shown in Figure 9. The second method of multiplication assumed that no GaAs multiplier could be found. Instead, the simple paper and pencil method of multiplying by shifting and adding is utilized. Multiplication based on the Booth algorithm was not investigated, but could possibly increase the speed of multiplication. See the Instruction Execution Timing section for the number of machine cycles required for each method.

The feedback latch, shown in Figure 10, is used to pass data from the LIDB to the UIDB. If data needs to be transferred from one ALU to another, the data can be moved to the feedback latch on one cycle and then transferred to the proper ALU on the next cycle. Besides just passing data from the LIDB to the UIDB, the feedback latch can perform two simple functions that help speed up some of the address mode calculations. A common element in many of the addressing modes is a data or address register that is used as an index

register and added to the base address of an operand's effective address. The index register can be either a long-word or a sign extended word scaled by 1, 2, 4, or 8. The size and scaling information for an index register is encoded in EWR bits 11-9. There are some control lines on the feedback latch that allow it to do both the sign extension and scaling. Sometimes the size of the sign extension is contained in the OP SIZE register. A multiplexer is used to enable either the PL, OP SIZE register, or bit 11 of the EWR to select the size of the sign extension. The PL can enable or disable the scaling circuit. When scaling is enabled, bits 10 and 9 of the EWR select the size of the scaling factor.

The shifter is used to emulate the eight shift functions of the MC68020. The shifter has two internal latches. One contains the data and the other contains the number of positions to shift the data. Both of these internal latches are loaded from the LIDB by enabling the DATA LOAD and SHIFT LOAD lines from the PL. After the data and shift latches are loaded, the PL supplies a 3-bit instruction that tells the shifter which shift operation to perform. The size of the shift operation is determined by the value in the OP SIZE register. Since some of the shift operations use the MC68020's X and C flags in the target status register, there is a control line that can load the X and C flags into the shifter. When the shift operation is done, the result can be read on the UIDB, and the resultant flags can be latched back into the target status register.

Miscellaneous Circuitry:

This section describes the circuitry which interfaces and supplements the sequencer and ALU sections of the host machine.

The main memory (MM) size multiplexer, shown in Figure 11, is used to select the source which indicates the size of the memory access to be performed.

The possible sources are the pipeline register, the OP SIZE register, and the EWR.

Two inputs to the CC multiplexer which have not yet been described are the Bcc circuit and the LIDB latch. The Bcc circuit, shown in Figure 12, uses the target level condition codes to generate the flags which the Bcc and Scc target level instructions use. Thus, the microsequencer can automatically test these flags without the ALU doing any computations. The LIDB latch, shown in Figure 6, simply latches the data on the LIDB each cycle. This allows the microsequencer to test each bit on the bus. In this manner, the microsequencer can test every bit of every device that can output to the LIDB.

There are five specialized circuits which gate data onto the UIDB. One gates the sign extended lower byte of the EWR onto the bus, allowing immediate access of this data, which helps several instructions such as Bcc. Similarly, another gates the sign extended EWR onto the bus. These two circuits are shown in Figures 13 and 14. Figure 15 shows a circuit which maps IR bits 11-9 to integers and gates them onto the bus. This helps instructions such as ADDQ. A PROM which has common constants for use in masking registers and shifting also can gate data onto the bus. The last specialized circuit, shown in Figure 6, is the OP SIZE offset PROM. It is used to support the postincrement and predecrement address register indirect subroutines. It takes as input the OP SIZE register and outputs a 1, 2, or 4, corresponding to byte, word, or long word. This allows the proper value to be added to or subtracted from the address register.

The final elements, shown in Figure 6, are those which support main memory accesses. The MAR is a 32-bit register which gates addresses onto the address bus. It is usually loaded from the address ALU via the LIDB. The DBI is used to read and write data to and from main memory. A sign extension

unit, controlled by the (MM) size multiplexer, sign extends all data read from main memory.

The Microinstruction Word:

Each microinstruction contains 55 fields and is a total of 123 bits wide. Since the memory used would probably be 128 bits wide, the remaining 5 bits could be used for enhancements in a future design. Appendix A contains a list of the fields of the microinstruction word and lists the name, number of bits, and "truth table" for each field. Since the microcode width has no impact on the final goals of this project, no effort has been made to minimize the width of the microinstruction word. However, through minimization procedures, like in reference 13, this could be reduced.

The Microprograms and Support Routines:

The first, and most obvious microprogram written for the host, was an instruction fetch routine called I_FETCH. The I_FETCH routine performs the following functions:

1.) Fetches a 16-bit instruction from the main memory pointed to by the PC, and loads it into the PIR

2.) Increments the PC by 2, reads in the word following the instruction, and stores it in the EWR

The word following the instruction may or may not be a valid extension word, but it was pre-fetched "for free" since it could be overlapped with the rest of I_FETCH. This is because we assumed that a GaAs main memory access could be completed in one cycle.[2] If this is not the case, the prefetch of the EWR could not be overlapped with the rest of the instruction fetch. Once the instruction is fetched, a JUMP MAP instruction is executed on the VS29G10A,

which causes the output of the IR PROM to be gated into the μPC. The IR PROM output is the starting address of the microprogram corresponding to each target level instruction.

When an instruction contains an effective address that is not data register direct or address register direct, a microsubroutine is called that will calculate the address of the operand in main memory. The starting address of the proper effective address microsubroutine is stored in the EA PROM. The inputs to the EA PROM are several control bits of the EWR and the 6-bit effective address stored in the EAR. To call a microsubroutine, a JUMP MAP is performed on the microsequencer and the output of the EA PROM is gated into the μPC. Hence, before performing a JUMP MAP, the EAR must be loaded with the proper effective address field from the IR. The effective address routines then calculate the effective address and place it in a register of the address ALU and also in the MAR. Thus, when an effective address routine returns to a calling routine, the calling routine only needs to do a main memory read to obtain the operand.

Timing Analysis:

In order to keep the critical path as short as possible, all inputs and outputs of the ALUs were latched. These latches separated the host machine into two sections, a microsequencer section and an ALU section. The worst case critical path for the microsequencer is shown in Figure 16. Table 4 shows the delays of each element in the path, using Si and GaAs parts. Similarly, Figure 17 and Table 5 show the critical path and element delays of the ALU section.

The interconnect delays were estimated by laying out the chips used in the critical path of both sections and assuming that signals propagate at 200 ps/inch. For glass epoxy microstrip, assuming a relative dielectric constant of

4.8, the delay is 150 ps/inch.[11] Due to the effects of cornering, impedance mismatches, multiple fanouts, and daisy chaining, we assumed an upper bound approximation of 200 ps/inch.

The critical paths listed in Tables 4 and 5 show that the microsequencer critical path and the ALU critical path are closely matched. As seen in Figure 17 and Table 5, the ALU is the only element in the ALU critical path. Thus, adding extra pipeline registers to this design has no effect on the cycle time. Since the ALU critical path is the longer of the two for GaAs, it determines the maximum clock frequency of the host, which is 34.66 MHz. For Si, the microsequencer critical path is the longer of the two, resulting in a maximum clock frequency of 9.17 MHz.

The interconnect delays add up to 11.1 percent of the ALU critical path for GaAs. If interconnect technology improves to the point where signals can propagate at the speed of light (the upper bound), the new maximum clock frequency of the host would be 37.03 MHz, a speed up of only 6%.

Instruction Execution Timing:

In order to determine the efficiency of this emulation of the MC68020, the number of cycles needed to emulate an instruction by the bit-slice host machine was compared to the number of cycles needed by the MC68020. This method was favored over timing benchmarks since benchmarks can be somewhat misleading. The time to execute a single MC68020 instruction is given by the formula:

$$\text{Execution time} = \frac{\#\text{ of machine cycles}}{\text{clock frequency}}$$

If the instruction mix of a specific application is known, then an estimate of the total execution time can be calculated by summing the execution times of all

the instructions.[14]

Since the host machine does not have an instruction cache and does not overlap the execution of two consecutive instructions, the number of cycles needed to emulate an instruction is independent of the surrounding instructions. The MC68020, however, does support these features, causing it to have cycle times that are dependent on the context in which the instruction is used. The best case for the MC68020 assumes that the instruction is in cache and the maximum possible overlap is achieved. The worst case for the MC68020 assumes that the instruction is not in cache, or the cache is disabled, and there is no instruction overlap. (For more information, see Chapter 9 of the *MC68020 32-Bit Microprocessor User's Manual.*) This makes a direct comparison difficult.

Another point that should be kept in mind when analyzing the speed of the emulation is the single cycle memory access of the host, which was mentioned earlier. This gives the GaAs bit slice machine an advantage on instructions and addressing modes which require several memory accesses. If this assumption seems unrealistic for a specific application, a worst case approximation for the number of cycles needed to implement an instruction or addressing mode can be calculated by adding two cycles for each memory access needed.

Tables 6-13 compare the number of cycles needed to fetch operands and execute instructions on the MC68020 and on the emulator. All of the times given for the MC68020 were taken from the *MC68020 32-Bit Microprocessor User's Manual*, and assume the following:

1.) All operands are long-word aligned, as is the stack
2.) 32-bit data bus
3.) No wait state memory (3 cycle read/write)

The times given for the emulator are based on the following assumptions:

1.) All operands are long-word aligned, as is the stack
2.) 32-bit data bus
3.) No wait state memory (1 cycle read/write)

As mentioned earlier, the user may wish to not use the GaAs hardware multiplier, due to cost or other trade-offs. In this event, Table 14 compares the number of cycles needed to do a multiply with and without a hardware multiplier. As indicated before, the multiply instruction implemented on the host machine without a hardware multiplier uses a simple shift and add algorithm.

The Simulation Program:

In order to debug design flaws in both the hardware and microcode, a software simulator was written in C. It can execute one or more microinstructions or target instructions, and allows a breakpoint to be set to facilitate debugging. Also, a trace mode can be enabled which prints the μPC as instructions are executed, as well as the number of microinstructions executed, so that the number of cycles of a microprogram can be easily obtained. Any register, target level main memory location, or μCS location can be viewed and modified. With this feature, the microprograms can be edited inside the simulator. The μCS and main memory are kept in separate files, and are read in at the beginning of each simulation.

The program, given in Appendix B, was modularly written in six separate files. The first file, *main.c*, contains the global structure definitions and global variable declarations. It initializes all variables and reads in the μCS and MM from external files. It also contains the main body of the simulator and miscellaneous subroutines. The file *change.c* contains the subroutines that change the state of the host. The file *menu.c* contains subroutines that print all of the

menus used by any other programming module. The file *shifter.c* contains the shifter subroutine. The file *prom.c* contains two subroutines. One subroutine emulates the EA PROM, and returns the starting addresses of the microroutines that perform effective address calculations. The other subroutine emulates the IR PROM, and returns the starting addresses of the microprograms which correspond to target level instructions. The file *execute.c* is the heart of the simulator. It simulates the execution of host and target level instructions, and emulates the functions of the host machine elements.

Conclusion:

As we have shown, GaAs bit slice components can be used in a microprogrammed machine to achieve a speed increase over silicon. However, three important points must be made. First, this machine has not actually been implemented in hardware, and the software that emulated the hardware is not able to predict timing problems that might exist if the actual machine were constructed. Second, several chips in this design are not commercially available and would need to be constructed using standard cell design. This would be quite costly in terms of both time and money. Finally, the overall cost of this project would be tremendous even without the custom chips, and in our opinion would far outweigh the benefits of the modest speed increases, unless an application requires components that are more environmentally hard than silicon devices.

During the design, several improvements to the machine were considered and suggested by colleagues, but not implemented. One improvement would be to design a custom ALU on GaAs which would have a slice larger than 4-bits and include CLA circuitry to facilitate quick additions. This would reduce the off-chip delays, and could possibly reduce the critical path length, thus speeding

up the machine.

Another possible improvement would be to model the MC68020 architecture instead of using the customary bit-slice design approach. This might allow the machine to enjoy the parallelism that the MC68020 employs. As an example, if a custom prefetch unit similar to the one in the MC68020 were used, several instructions could be fetched at once, allowing instructions to be overlapped. Thus, some instructions could be executed in an effective time of zero machine cycles.

While the cost of using a bit slice approach to CISC emulation is quite large, it does provide a means to implement a CISC using all GaAs parts, providing a system with environmental operating ranges superior to silicon.

References

1. L. Byington and D. Theis, "Air Force Standard 1750A ISA is the New Trend," *Computer*, November 1986, pp. 50-59.

2. V. Milutinovic and D. Fura, "An Introduction to GaAs Microprocessor Architecture for VLSI," *Computer*, March 1986, pp. 30-41.

3. V. Milutinovic, "GaAs Microprocessor Technology," *Computer*, October 1986, pp. 10-13.

4. K. McNeley and V. Milutinovic, "Emulating a Complex Instruction Set Computer with a Reduced Instruction Set Computer," *IEEE Micro*, February 1987, pp. 60-71.

5. Motorola, Inc., *MC68020 32-bit Microprocessor User's Manual*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

6. L. Ciminiera and A. Valenzano, *Advanced Microprocessor Architectures*, Addison-Wesley Publishing Co., 1987, pp. 231-282.

7. Vitesse Semiconductor Corporation, VS29G01 and VS29G10A Data Sheets, 741 Calle Plano, Camarillo, California, 93010.

8. D. Siewiorek, C. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill Book Company, 1982, pp. 168-211.

9. Y. Nakayama et. al., "A GaAs 16 x 16 bit Parallel Multiplier," *IEEE Journal of Solid-State Circuits*, October 1983, pp. 599-603

10. T. Furutsuka et. al., "A GaAs 12 x 12 bit Expandable Parallel Multiplier LSI Using Sidewall-Assisted Closely- Spaced Electrode Technology", *IEDM 84*, pp. 344-347.

11. C. Deierling, *Guidelines for the Use of Digital GaAs ICs*, GigaBit Logic Inc., 1908 Oak Terrace Lane, Newbury Park, CA 91320-5524, January 1987.

12. F. Lee et. al., "A High-Speed GaAs 8 x 8 bit Parallel Multiplier", *IEEE Journal of Solid-State Circuits*, August 1982, pp. 638-649.

13. S. Das et. al., "On Control Memory Minimization in Microprogrammed Digital Computers," *IEEE Transactions on Computers*, September 1973, pp. 845-848.

14. D. Ferrari, *Computer Systems Performance Evaluation*, Prentice-Hall Inc., 1978.

List of Tables

Table	Title	Reference
1	ALU Source Operand Control (From VS29G01 Data Sheet, Vitesse Semiconductor Corporation, 741 Calle Plano, Camarillo, California, 93010, January 1988, Figure 3. © 1988 Vitesse Semiconductor Corp.)	7
2	ALU Function Control (From VS29G01 Data Sheet, Vitesse Semiconductor Corporation, 741 Calle Plano, Camarillo, California, 93010, January 1988, Figure 4. © 1988 Vitesse Semiconductor Corp.)	7
3	ALU Destination Control (From VS29G01 Data Sheet, Vitesse Semiconductor Corporation, 741 Calle Plano, Camarillo, California, 93010, January 1988, Figure 5. © 1988 Vitesse Semiconductor Corp.)	7
4	Microsequencer Critical Path Timing Analysis	15
5	ALU Critical Path Timing Analysis	15
6	Cycle Times for Fetching Effective Addresses	15
7	Cycle Times for Calculating Effective Addresses	15
8	Cycle Times for Move Instructions	15
9	Cycle Times for Arithmetic/Logical Operations	15
10	Cycle Times for Single Operand Instructions	15
11	Cycle Times for Shift/Rotate Instructions	15
12	Cycle Times for Conditional Branch Instructions	15
13	Cycle Times for Control Instructions	15
14	Cycle Times for Firmware and Hardware Multiply	15

Table 1

Mnemonic	MICRO CODE				ALU SOURCE OPERANDS	
	I_2	I_1	I_0	Octal Code	R	S
AQ	L	L	L	0	A	Q
AB	L	L	H	1	A	B
ZQ	L	H	L	2	O	Q
ZB	L	H	H	3	O	B
ZA	H	L	L	4	O	A
DA	H	L	H	5	D	A
DQ	H	H	L	6	D	Q
DZ	H	H	H	7	D	O

Table 2

Mnemonic	MICRO CODE				ALU Function	Symbol
	I_2	I_1	I_0	Octal Code		
ADD	L	L	L	0	R Plus S	R + S
SUBR	L	L	H	1	S Minus R	S - R
SUBS	L	H	L	2	R Minus S	R - S
OR	L	H	H	3	R OR S	R $\vee$ S
AND	H	L	L	4	R AND S	R $\wedge$ S
NOTRS	H	L	H	5	$\overline{R}$ AND S	$\overline{R}$ $\wedge$ S
EXOR	H	H	L	6	R EX-OR S	R $\bar{\vee}$ S
EXNOR	H	H	H	7	R EX-NOR S	$\overline{R \bar{\vee} S}$

Table 3

Mnemonic	MICRO CODE				RAM FUNCTION		Q-REG. FUNCTION		Y OUTPUT	RAM SHIFTER		Q SHIFTER	
	I_2	I_1	I_0	Octal Code	Shift	Load	Shift	Load		RAM_0	RAM_3	Q_0	Q_3
QREG	L	L	L	0	X	NONE	NONE	F→Q	F	X	X	X	X
NOP	L	L	H	1	X	NONE	X	NONE	F	X	X	X	X
RAMA	L	H	L	2	NONE	F→B	X	NONE	A	X	X	X	X
RAMF	L	H	H	3	NONE	F→B	X	NONE	F	X	X	X	X
RAMQD	H	L	L	4	DOWN	F/2→B	DOWN	Q/2→Q	F	F_0	IN_0	Q_0	IN_3
RAMD	H	L	H	5	DOWN	F/2→B	X	NONE	F	F_0	IN_0	Q_0	X
RAMQU	H	H	L	6	UP	2F→B	UP	2Q→Q	F	IN_0	F_0	IN_0	Q_3
RAMU	H	H	H	7	UP	2F→B	X	NONE	F	IN_0	F_0	X	Q_3

X = Don't care. Electrically, the shift pin is an ECL input internally connected to an output in the high-impedance state.
B = Register addressed by B inputs.
Up is toward MSB, DOWN is toward LSB.

Table 4

Path Element	Si (nsec)	GaAs (nsec)
Register (CLK → output)	10.50	0.95
5 inch interconnect	1.00	1.00
CC mux (input → output)	15.70	2.16
1 inch interconnect	0.20	0.20
XOR gate	7.00	0.56
1 inch interconnect	0.20	0.20
Microsequencer (CC → output)	30.00	9.40
18 inch interconnect	3.60	3.60
uCS	35.00	4.10
18 inch interconnect	3.60	3.60
Register set-up time	2.00	0.50
TOTAL	109.00	26.27

Table 5

Path Element	Si (nsec)	GaAs (nsec)
Register (CLK → output)	10.50	0.95
5 inch interconnect	1.00	1.00
8:1 mux (input → output)	4.50	0.60
3 inch interconnect	0.60	0.60
ALU (A,B bus → G,P)	37.00	12.00
2 inch interconnect	0.40	0.40
CLA	21.40	2.00
2 inch interconnect	0.40	0.40
ALU (Cin → F≠0)	25.00	9.00
3 inch interconnect	0.60	0.60
4:1 mux (input → output)	4.50	0.60
1 inch interconnect	0.20	0.20
Register set-up time	2.00	0.50
TOTAL	108.10	28.85

Table 6

Address Mode	MC68020 Best	MC68020 Worst	Emulator Best	Emulator Worst
Dn	0	0	0	0
An	0	0	0	0
(An)	3	4	2	2
(An)+	4	4	3	3
-(An)	3	5	3	3
(d16,An) or (d16,PC)	3	6	3	3
(xxx).W	3	6	3	3
(xxx).L	3	7	4	4
#<data>.B	0	3	2	2
#<data>.W	2	3	2	2
#<data>.L	1	5	2	2
(d8,An,Xn) or (d8,PC,Xn)	4	8	4	5
(d16,An,Xn) or (d16,PC,Xn)	4	9	5	6
(B)	4	9	2	6
(d16,B)	6	12	4	6
(d32,B)	10	16	4	6
([B],I)	9	13	4	6
([B],I,d16)	11	16	6	8
([B],I,d32)	11	17	6	8
([d16,B],I)	11	16	6	8
([d16,B],I,d16)	13	19	8	10
([d16,B],I,d32)	13	20	8	10
([d32,B],I)	15	20	6	8
([d32,B],I,d16)	17	22	8	10
([d32,B],I,d32)	17	24	8	10

B = Base address; 0, An, PC, Xn, An+Xn, PC+Xn

I = Index; 0, Xn

NOTES: Xn cannot be in B and I at the same time.
Scaling and size of Xn does not affect timing.

Table 7

Address Mode	MC68020 Best	MC68020 Worst	Emulator Best	Emulator Worst
Dn	0	0	0	0
An	0	0	0	0
(An)	2	2	1	1
(An)+	2	2	2	2
-(An)	2	2	2	2
(d16,An) or (d16,PC)	2	3	2	2
#<data>.W	2	3	2	2
#<data>.L	1	5	3	3
(d8,An,Xn) or (d8,PC,Xn)	1	5	3	4
(d16,An,Xn) or (d16,PC,Xn)	3	7	4	5
(B)	3	7	1	5
(d16,B)	5	10	3	5
(d32,B)	9	15	3	5
([B],I)	8	12	3	5
([B],I,d16)	10	15	5	7
([B],I,d32)	10	16	5	7
([d16,B],I)	10	15	5	7
([d16,B],I,d16)	12	18	7	9
([d16,B],I,d32)	12	19	7	9
([d32,B],I)	14	19	5	7
([d32,B],I,d16)	16	21	7	9
([d32,B],I,d32)	16	24	7	9

B = Base address; 0, An, PC, Xn, An+Xn, PC+Xn

I = Index; 0, Xn

NOTES: Xn cannot be in B and I at the same time.
Scaling and size of Xn does not affect timing.

Table 8

	Instruction		MC68020 Best	MC68020 Worst	Emulator Best	Emulator Worst
	MOVE	Dn,Dn	0	3	2	2
	MOVE	An,Dn	0	3	3	3
*	MOVE	<ea>,Dn	#	#	4	4
*	MOVE	Dn,<ea>	#	#	6	6
	MOVE	An,<ea>	#	#	6	6
	MOVE	<ea>,<ea>	#	#	8	8
	MOVEA	Dn,An	0	3	3	3
	MOVEA	An,An	0	3	3	3
	MOVEA	<ea>,An	#	#	4	4
	MOVE	CCR,Dn	4	5	2	2
*	MOVE	CCR,<ea>	4	5	5	5
	MOVE	Dn,CCR	4	5	3	3
*	MOVE	<ea>,CCR	4	5	4	4
	MOVEQ	#<data>,Dn	0	3	2	2

* Add Fetch Effective Address time to MC68020 cases.
Add Calculate Effective Address time to the emulation cases.

\# See *MC68020 32-Bit Microprocessor User's Manual*, pp. 9-15 to 9-20.

Table 9

	Instruction		MC68020 Best	MC68020 Worst	Emulator Best	Emulator Worst
	ADD	Dn,Dn	0	3	3	3
*	ADD	<ea>,Dn	0	3	4	4
*	ADD	Dn,<ea>	3	6	5	5
*	ADDA	<ea>,An	0	3	5	5
	AND	Dn,Dn	0	3	3	3
*	AND	<ea>,Dn	0	3	4	4
*	AND	Dn,<ea>	3	6	5	5
	EOR	Dn,Dn	0	3	3	3
*	EOR	Dn,<ea>	3	6	5	5
	OR	Dn,Dn	0	3	3	3
*	OR	<ea>,Dn	0	3	4	4
*	OR	Dn,<ea>	3	6	5	5
	SUB	Dn,Dn	0	3	3	3
*	SUB	<ea>,Dn	0	3	4	4
*	SUB	Dn,<ea>	3	6	5	5
*	SUBA	<ea>,An	0	3	5	5
	CMP	Dn,Dn	0	3	2	2
	CMP	An,Dn	0	3	3	3
*	CMP	<ea>,Dn	0	3	4	4
	CMPA	Dn,An	1	4	3	3
	CMPA	An,An	1	4	2	2
*	CMPA	<ea>,An	1	4	4	4
	MUL.W	Dn,Dn	25	28	10	10
*	MUL.W	<ea>,Dn	25	28	10	10
	MUL.L	Dn,Dl	41	44	11	12
*	MUL.L	<ea>,Dl	41	44	14	15
	MUL.L	Dn,Dh:Dl	41	44	10	10
*	MUL.L	<ea>,Dh:Dl	41	44	13	13
	DIVS.W	Dn,Dn	54	56	50	52
*	DIVS.W	<ea>,Dn	54	56	51	53
	DIVS.L	Dn,Dn	88	90	78	80
*	DIVS.L	<ea>,Dq	88	90	79	81
	DIVS.L	Dn,Dr:Dq	88	90	78	83
*	DIVS.L	<ea>,Dr:Dq	88	90	80	85
	DIVSL.L	Dn,Dr:Dq	88	90	78	80
*	DIVSL.L	<ea>,Dr:Dq	88	90	79	81
	DIVU.W	Dn,Dn	42	44	48	48
*	DIVU.W	<ea>,Dn	42	44	48	48
	DIVU.L	Dn,Dq	76	78	73	73
*	DIVU.L	<ea>,Dq	76	78	75	75
	DIVU.L	Dn,Dr:Dq	76	78	75	75
*	DIVU.L	<ea>,Dr:Dq	76	78	77	77
	DIVUL.L	Dn,Dr:Dq	76	78	73	73
*	DIVUL.L	<ea>,Dr:Dq	76	78	75	75

* Add Fetch Effective Address time to MC68020 cases.
Add Calculate Effective Address time to the emulation cases.

Table 10

	Instruction		MC68020 Best	MC68020 Worst	Emulator Best	Emulator Worst
	CLR	Dn	0	3	2	2
#	CLR	<ea>	3	6	5	5
	NEG	Dn	0	3	2	2
*	NEG	<ea>	3	6	6	6
	NOT	Dn	0	3	2	2
*	NOT	<ea>	3	6	6	6
	EXT	Dn	1	4	3	3
	Scc	Dn	1	4	2	2
#	Scc	<ea>	6	6	6	6
	TST	Dn	0	3	2	2
*	TST	<ea>	0	3	4	4

* Add Fetch Effective Address time to MC68020 cases.
Add Calculate Effective Address time to the emulation cases.

\# Add Calculate Effective Address time to MC68020 cases.
Add Calculate Effective Address time to the emulation cases.

Table 11

	Instruction		MC68020 Best	MC68020 Worst	Emulator Best	Emulator Worst
	LSL	Dn,Dn	3	6	5	5
*	LSL	Mem by 1	5	6	6	6
	LSR	Dn,Dn	3	6	5	5
*	LSR	Mem by 1	5	6	6	6
	ASL	Dn,Dn	5	8	6	6
*	ASL	Mem by 1	6	7	7	7
	ASR	Dn,Dn	3	6	6	6
*	ASR	Mem by 1	5	6	7	7
	ROL	Dn,Dn	5	8	5	5
*	ROL	Mem by 1	7	7	6	6
	ROR	Dn,Dn	5	8	5	5
*	ROR	Mem by 1	7	7	6	6
	ROXL	Dn,Dn	9	12	5	5
*	ROXL	Mem by 1	5	6	6	6
	ROXR	Dn,Dn	9	12	5	5
*	ROXR	Mem by 1	5	6	6	6

* Add Fetch Effective Address time to MC68020 cases.
Add Calculate Effective Address time to the emulation cases.

Table 12

Instruction		MC68020 Best	MC68020 Worst	Emulator Best	Emulator Worst
Bcc	(taken)	3	9	4	4
Bcc.B	(not taken)	1	5	2	2
Bcc.W	(not taken)	3	7	4	4
Bcc.L	(not taken)	3	9	4	4
DBcc	(cc = false,count > -1)	3	9	3	3
DBcc	(cc = false,count = -1)	7	10	6	6
DBcc	(cc = true)	3	7	3	3

Table 13

	Instruction		MC68020 Best	MC68020 Worst	Emulator Best	Emulator Worst
	BSR.B		5	13	7	7
	BSR.W		5	13	7	7
	BSR.L		5	13	8	8
	CHK	Dn,Dn	8	8	4	4
*	CHK	<ea>,Dn	8	8	6	6
**	CHK2	<ea>,Dn	16	18	9	12
**	CHK2	<ea>,An	16	18	10	13
%	JMP	<ea>	1	7	5	5
%	JSR	<ea>	3	11	8	8
#	LEA	<ea>,An	2	3	5	5
	NOP		2	3	2	2
#	PEA	<ea>	3	6	6	6
	RTD	#<displacement>	9	12	5	5
	RTR		13	15	6	6
	RTS		9	12	5	5

* Add Fetch Effective Address time to MC68020 cases.
Add Calculate Effective Address time to the emulation cases.

% See *MC68020 32-Bit Microprocessor User's Manual*, pp. 9-15.

** See *MC68020 32-Bit Microprocessor User's Manual*, pp. 9-12.

Table 14

	Instruction		Firmware Best	Firmware Worst	Hardware Best	Hardware Worst
	MUL.W	Dn,Dn	21	37	10	10
*	MUL.W	<ea>,Dn	22	38	10	10
	MUL.L	Dn,Dl	22	38	11	12
*	MUL.L	<ea>,Dl	24	40	14	15
	MUL.L	Dn,Dh:Dl	39	103	10	10
*	MUL.L	<ea>,Dh:Dl	41	105	13	13

* Add Calculate Effective Address time.

List of Figures

Figure	Title	Reference
1	MC68020 Block Diagram (From MC68020 32-Bit Microprocessor User's Manual, Prentice-Hall, Englewood Cliffs, NJ, 07632, Figure 1-1. © 1985 Motorola Inc.)	5
2	User Programming Model (From MC68020 32-Bit Microprocessor User's Manual, Prentice-Hall, Englewood Cliffs, NJ, 07632, Figure 1-2. © 1985 Motorola Inc.)	5
3	VE29G01 Block Diagram (From VS29G01 Data Sheet, Vitesse Semiconductor Corporation, 741 Calle Plano, Camarillo, California, 93010, January 1988, pg. 1. © 1988 Vitesse Semiconductor Corp.)	7
4	VE29G10A Block Diagram (From VS29G01 Data Sheet, Vitesse Semiconductor Corporation, 741 Calle Plano, Camarillo, California, 93010, January 1988, pg. 1. © 1988 Vitesse Semiconductor Corp.)	7
5	Control Unit Section	15
6	ALU Section	15
7	Address ALU	15
8	Data ALU	15
9	Hardware Multiplier	15
10	Feedback Latch	15
11	Memory Interface Support Circuitry	15
12	Bcc Circuitry	15
13	EWR Bits 7-0 Sign Extension	15
14	EWR Bits 15-0 Sign Extension	15
15	IR bits 11-9 Mapping Circuit	15
16	Microsequencer Critical Path	15
17	ALU Section Critical Path	15

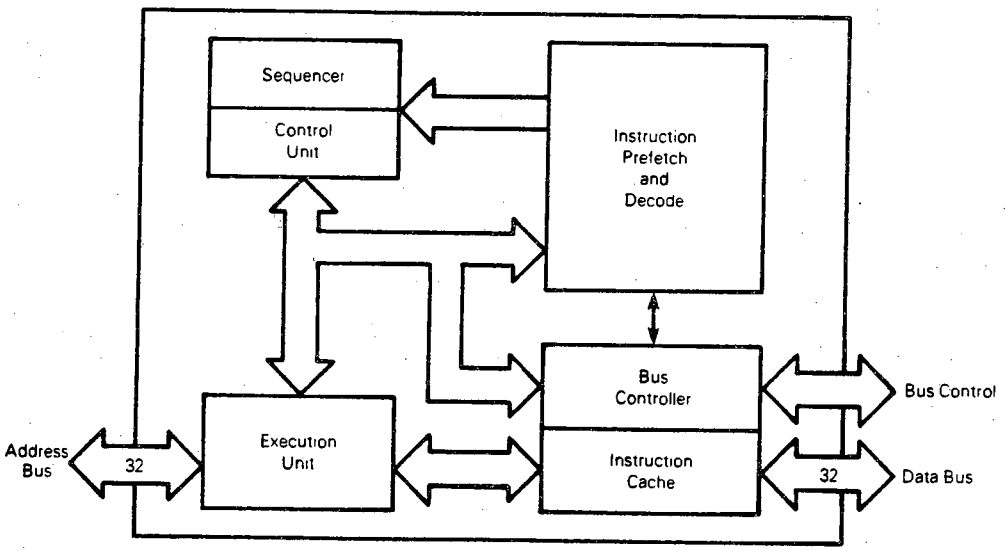

Sequencer
Control Unit
Instruction Prefetch and Decode
Bus Controller
Bus Control
Address Bus
32
Execution Unit
Instruction Cache
32
Data Bus

Figure 1

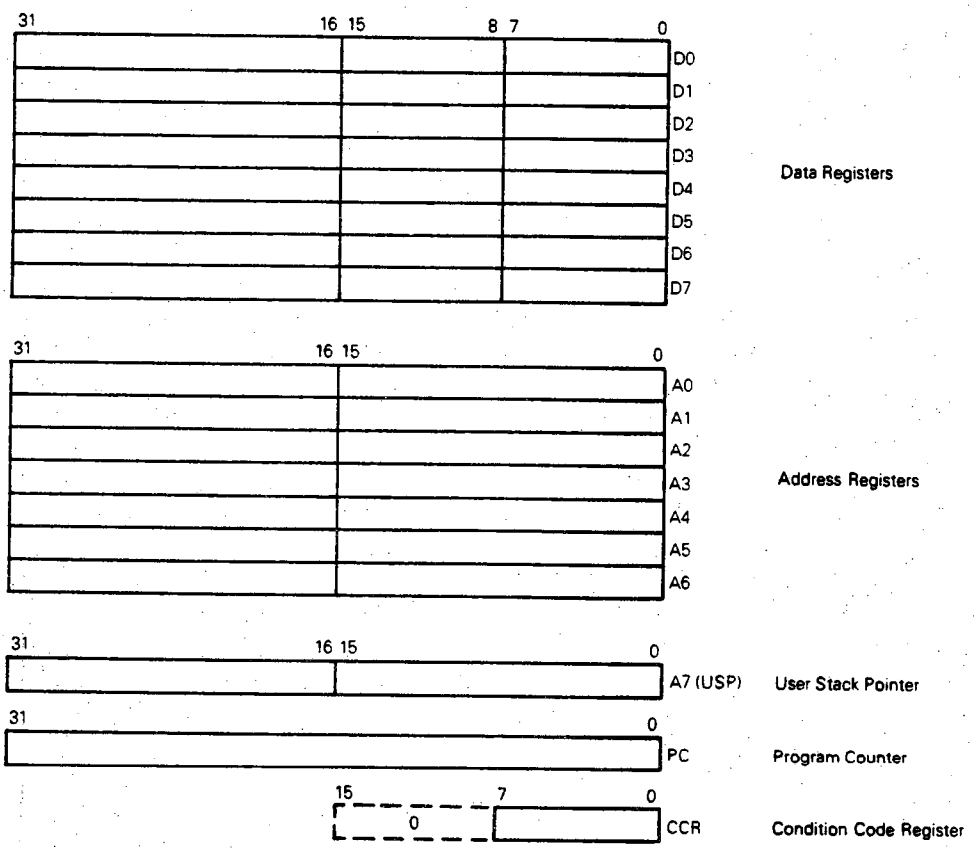

31
16 15
8 7
0
D0
D1
D2
D3
D4
D5
D6
D7
Data Registers
31
16 15
0
A0
A1
A2
A3
A4
A5
A6
Address Registers
31
16 15
0
A7 (USP)
User Stack Pointer
31
0
PC
Program Counter
15
7
0
0
CCR
Condition Code Register

Figure 2

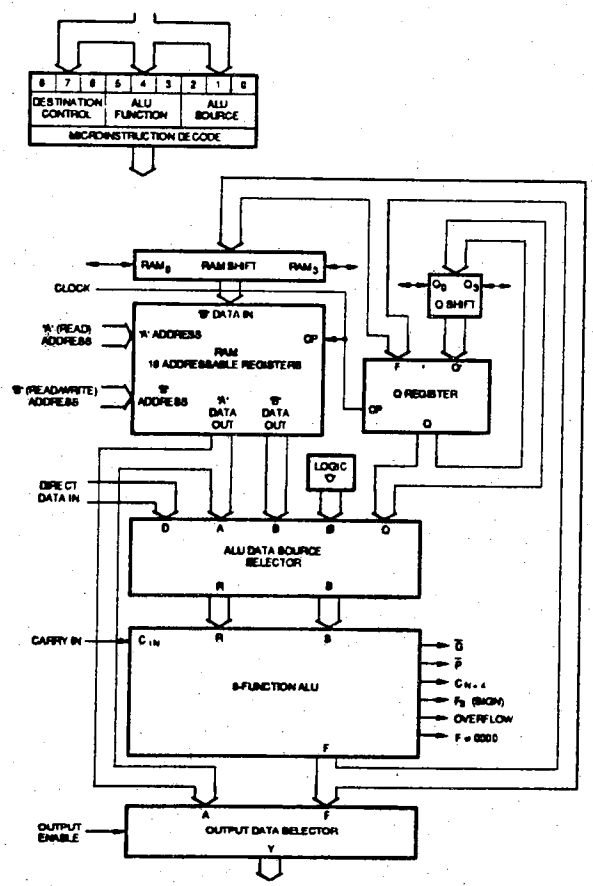

DESTINATION CONTROL
ALU FUNCTION
ALU SOURCE
MICROINSTRUCTION DECODE
CLOCK
DATA IN
'A' (READ) ADDRESS
'A' ADDRESS
RAM
16 ADDRESSABLE REGISTERS
'B' (READ/WRITE) ADDRESS
'B' ADDRESS
'A' DATA OUT
'B' DATA OUT
Q SHIFT
Q REGISTER
LOGIC
DIRECT DATA IN
ALU DATA SOURCE SELECTOR
CARRY IN
8-FUNCTION ALU
OVERFLOW
OUTPUT ENABLE
OUTPUT DATA SELECTOR

Figure 3

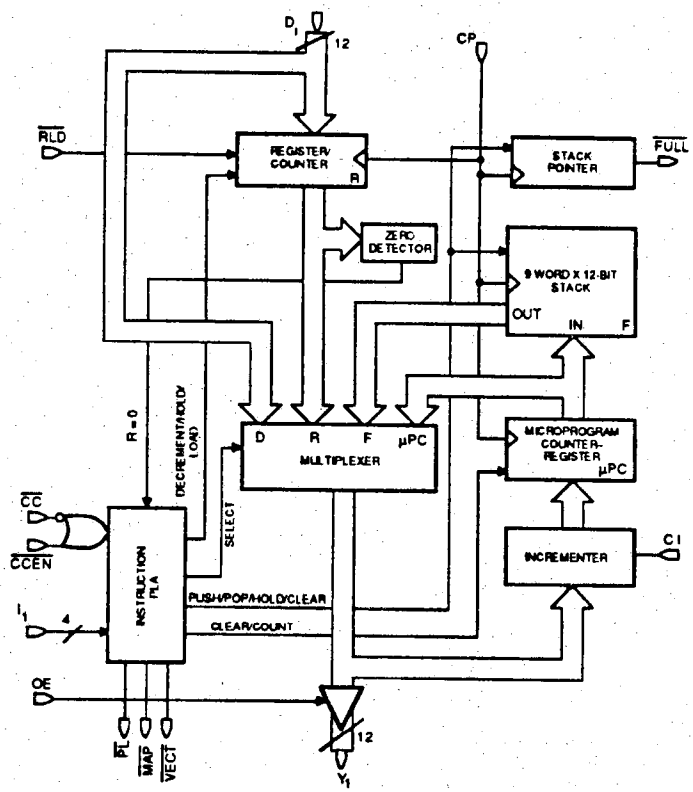

D1
12
CP
RLD
REGISTER/ COUNTER
R
STACK POINTER
FULL
ZERO DETECTOR
9 WORD X 12-BIT STACK
OUT
IN
F
R = 0
DECREMENT/HOLD/ (LOAD)
D
R
F
µPC
MULTIPLEXER
MICROPROGRAM COUNTER-REGISTER
µPC
SELECT
CC
CCEN
INSTRUCTION PLA
INCREMENTER
CI
I1
4
PUSH/POP/HOLD/CLEAR
CLEAR/COUNT
OE
PL
MAP
VECT
12
Y1

Figure 4

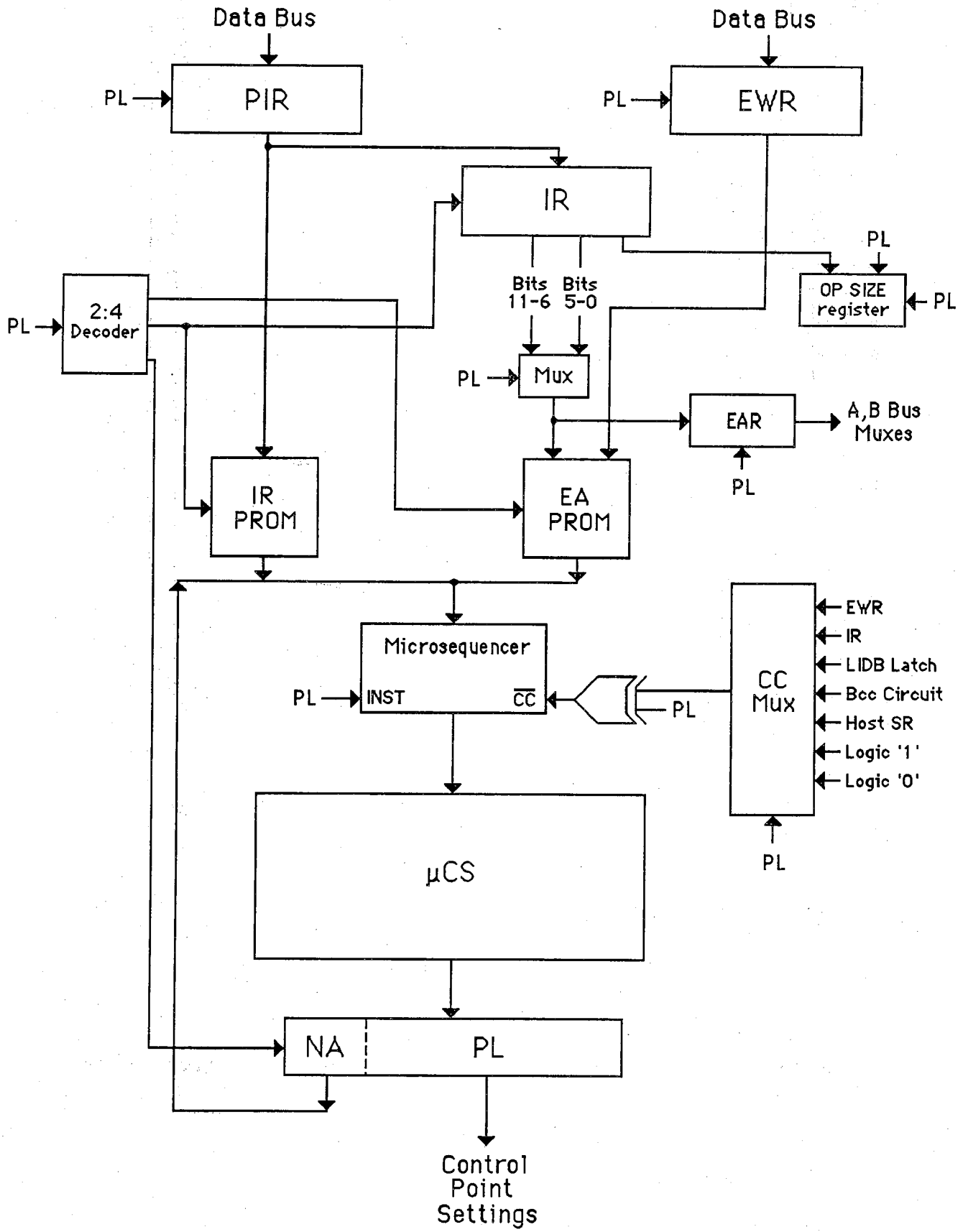

Data Bus
Data Bus
PL
PIR
PL
EWR
IR
PL
Bits 11-6
Bits 5-0
OP SIZE register
PL
PL
2:4 Decoder
PL
Mux
EAR
A,B Bus Muxes
PL
IR PROM
EA PROM
Microsequencer
PL
INST
CC
PL
CC Mux
EWR
IR
LIDB Latch
Bcc Circuit
Host SR
Logic '1'
Logic '0'
PL
µCS
NA
PL
Control Point Settings

Figure 5

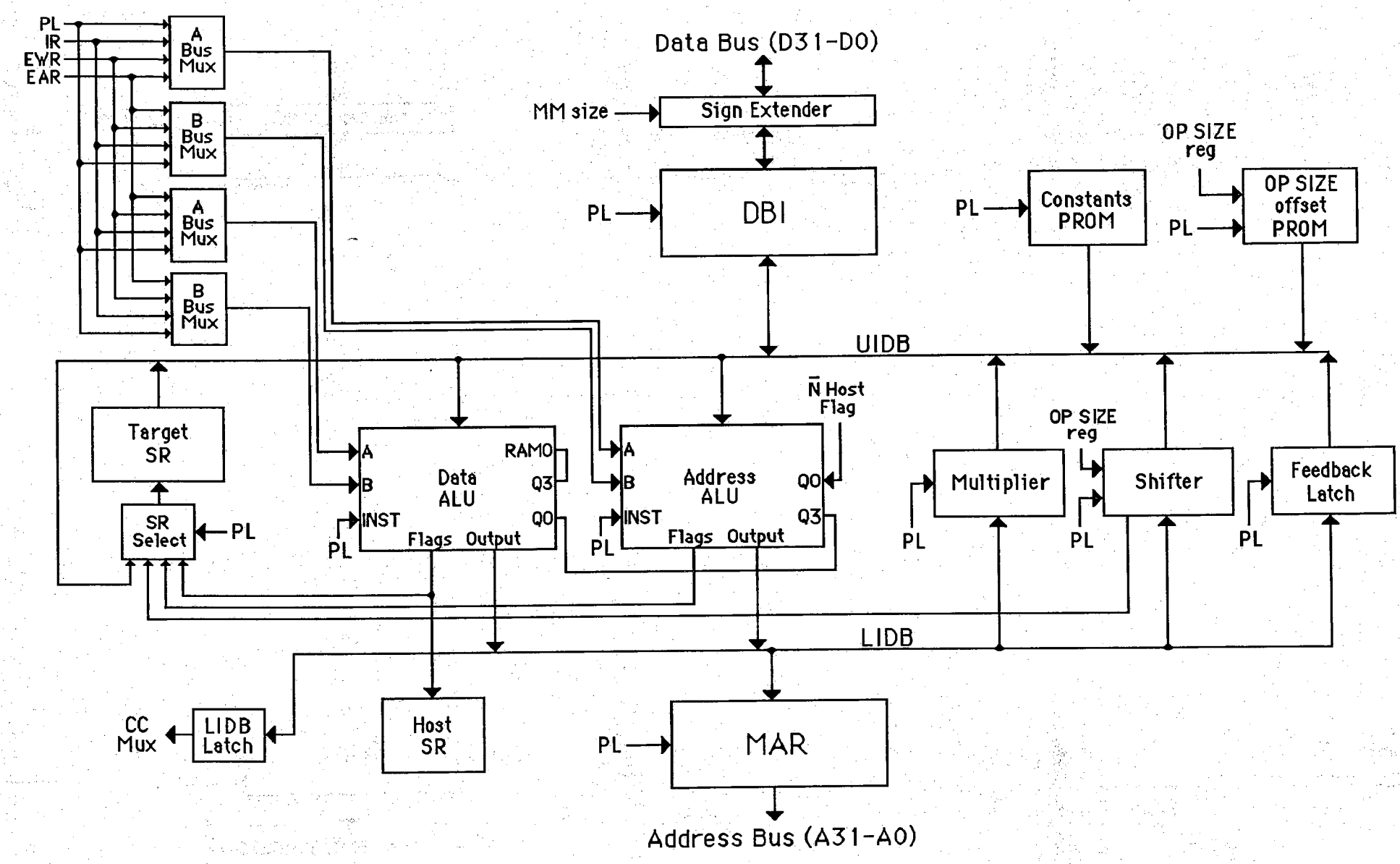

PL
IR
EWR
EAR
A Bus Mux
B Bus Mux
A Bus Mux
B Bus Mux
Data Bus (D31-D0)
MM size
Sign Extender
PL
DBI
PL
Constants PROM
OP SIZE reg
PL
OP SIZE offset PROM
UIDB
Target SR
SR Select
PL
A
B
INST
Data ALU
RAM0
Q3
Q0
Flags Output
PL
A
B
INST
Address ALU
Q0
Q3
Flags Output
PL
N̄ Host Flag
PL
Multiplier
OP SIZE reg
PL
Shifter
PL
Feedback Latch
LIDB
CC Mux
LIDB Latch
Host SR
PL
MAR
Address Bus (A31-A0)

Figure 6

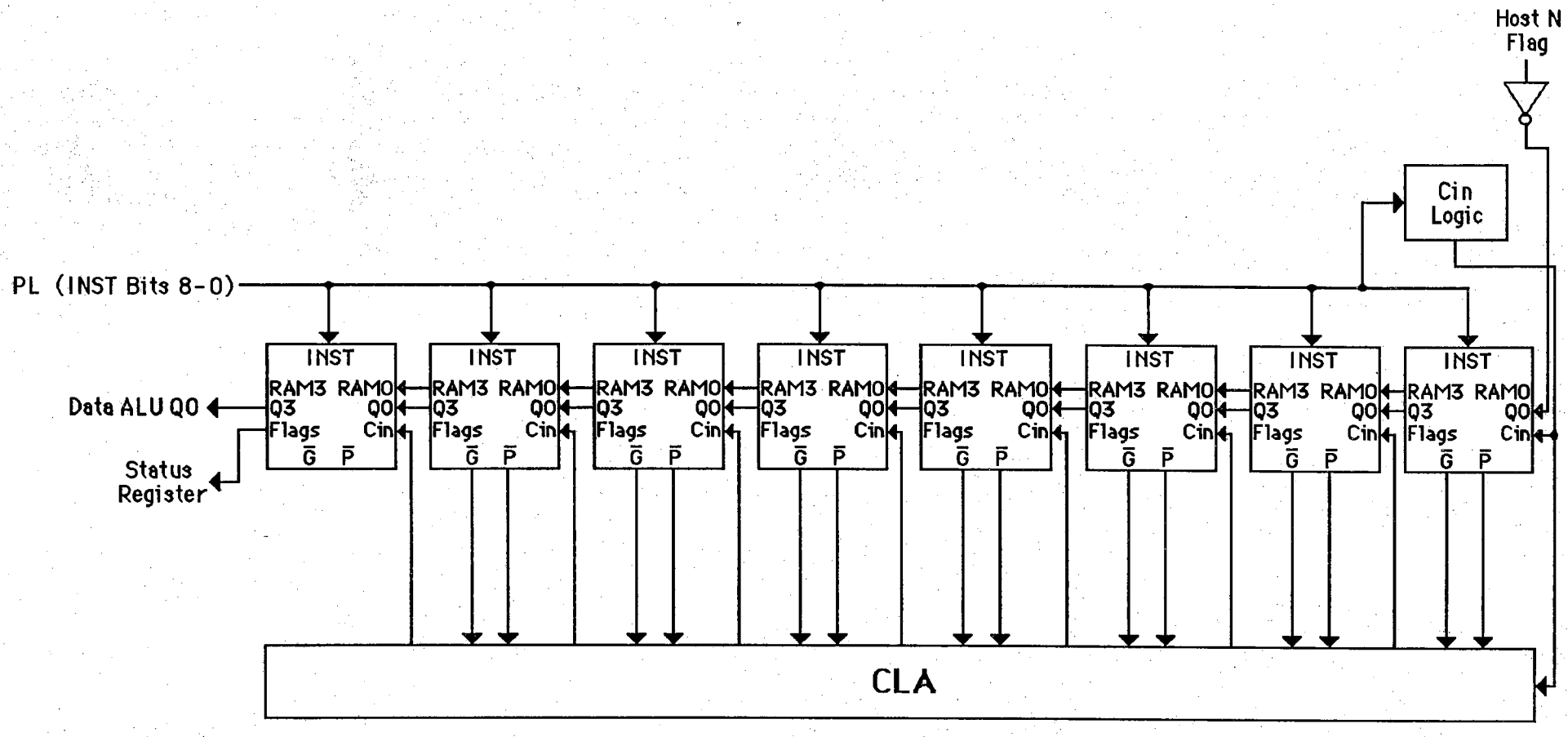

Host N Flag
Cin Logic
PL (INST Bits 8-0)
INST
RAM3 RAM0
Q3 Q0
Flags Cin
G P
Data ALU Q0
Status Register
CLA

Figure 7

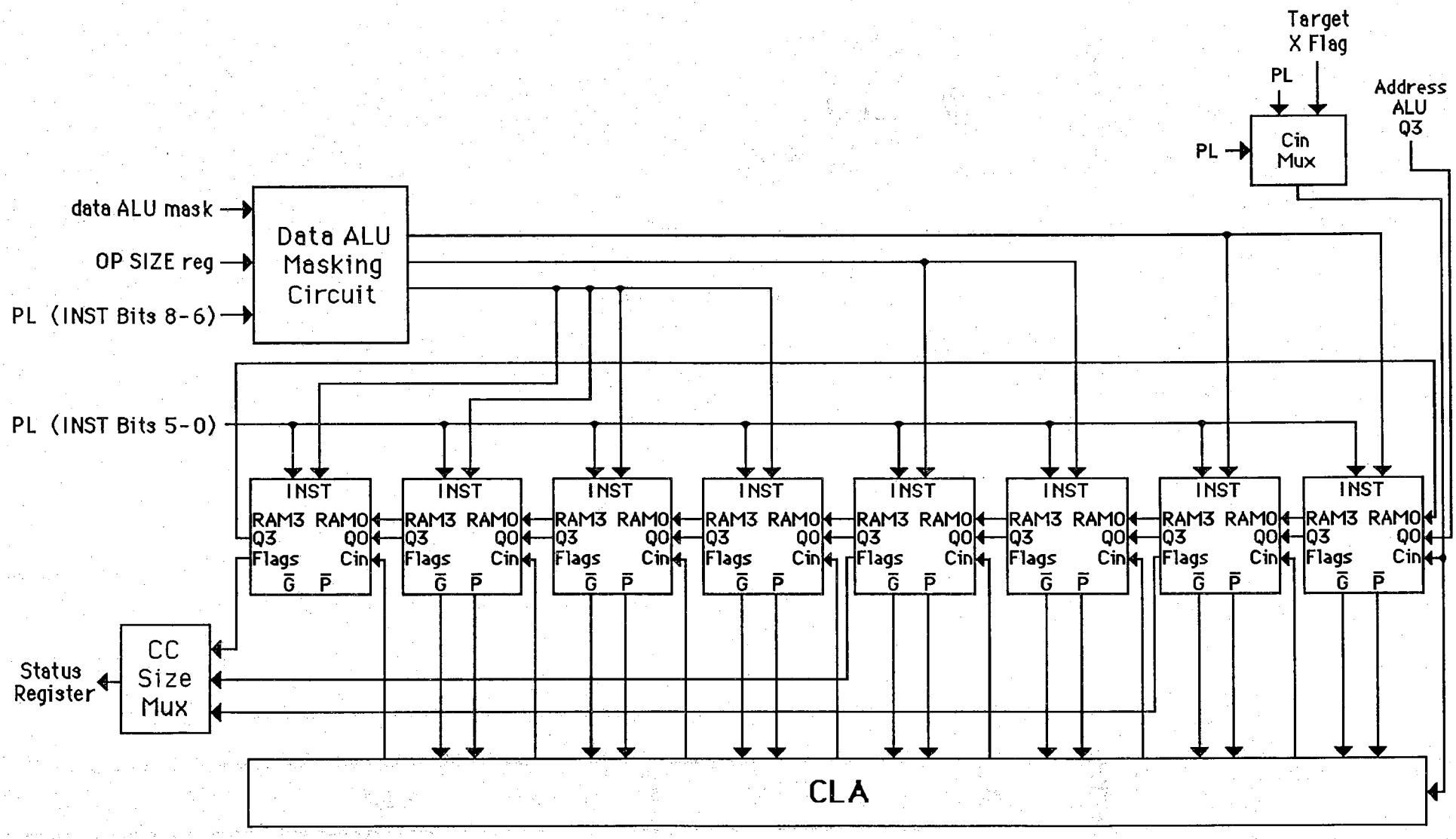

Target
X Flag
PL
Address
ALU
Q3
PL
Cin
Mux
data ALU mask
OP SIZE reg
PL (INST Bits 8-6)
Data ALU
Masking
Circuit
PL (INST Bits 5-0)
INST
RAM3 RAM0
Q3 Q0
Flags Cin
G P
Status
Register
CC
Size
Mux
CLA

Figure 8

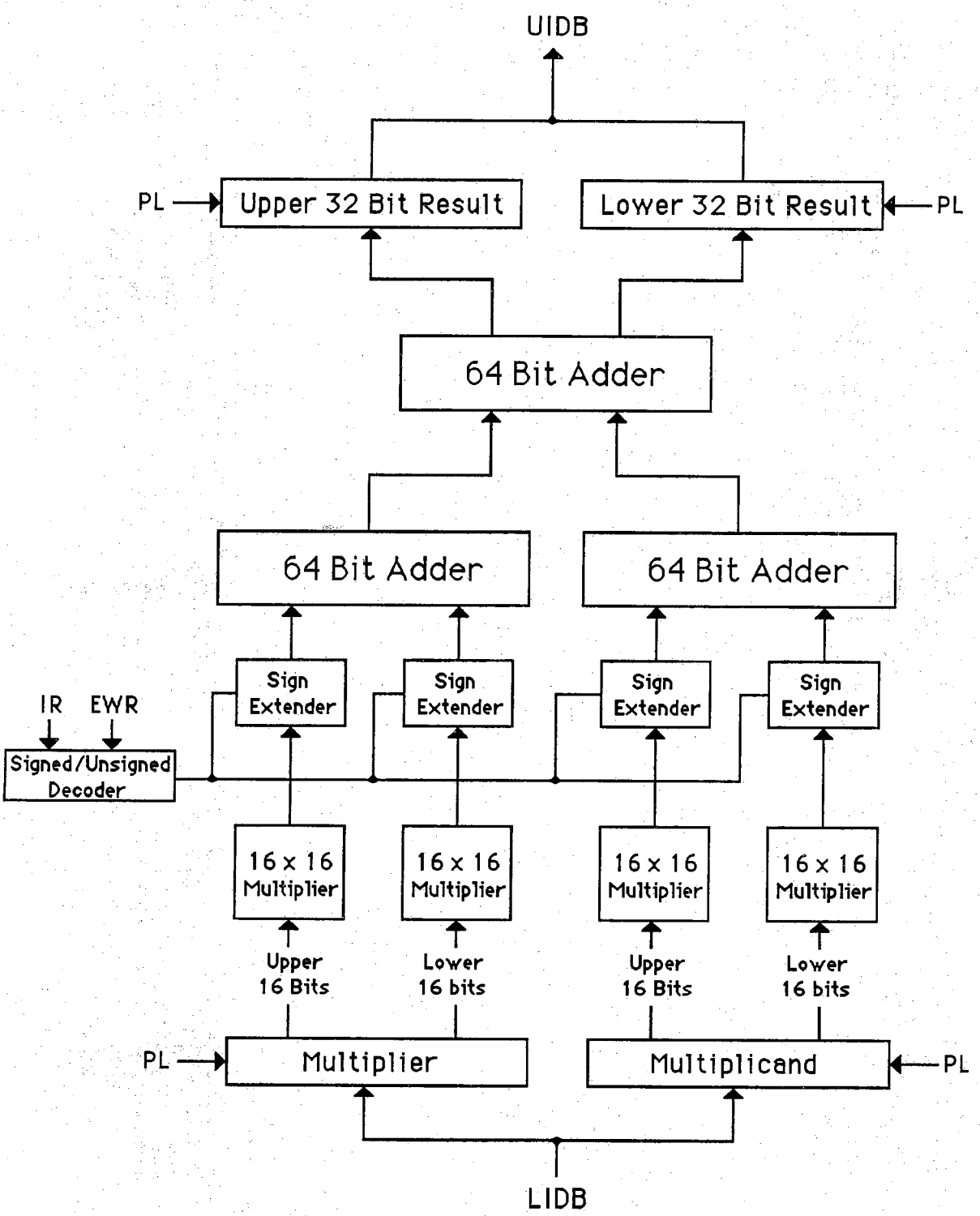

UIDB
PL
Upper 32 Bit Result
Lower 32 Bit Result
PL
64 Bit Adder
64 Bit Adder
64 Bit Adder
IR
EWR
Signed/Unsigned Decoder
Sign Extender
Sign Extender
Sign Extender
Sign Extender
16 x 16 Multiplier
16 x 16 Multiplier
16 x 16 Multiplier
16 x 16 Multiplier
Upper 16 Bits
Lower 16 bits
Upper 16 Bits
Lower 16 bits
PL
Multiplier
Multiplicand
PL
LIDB

Figure 9

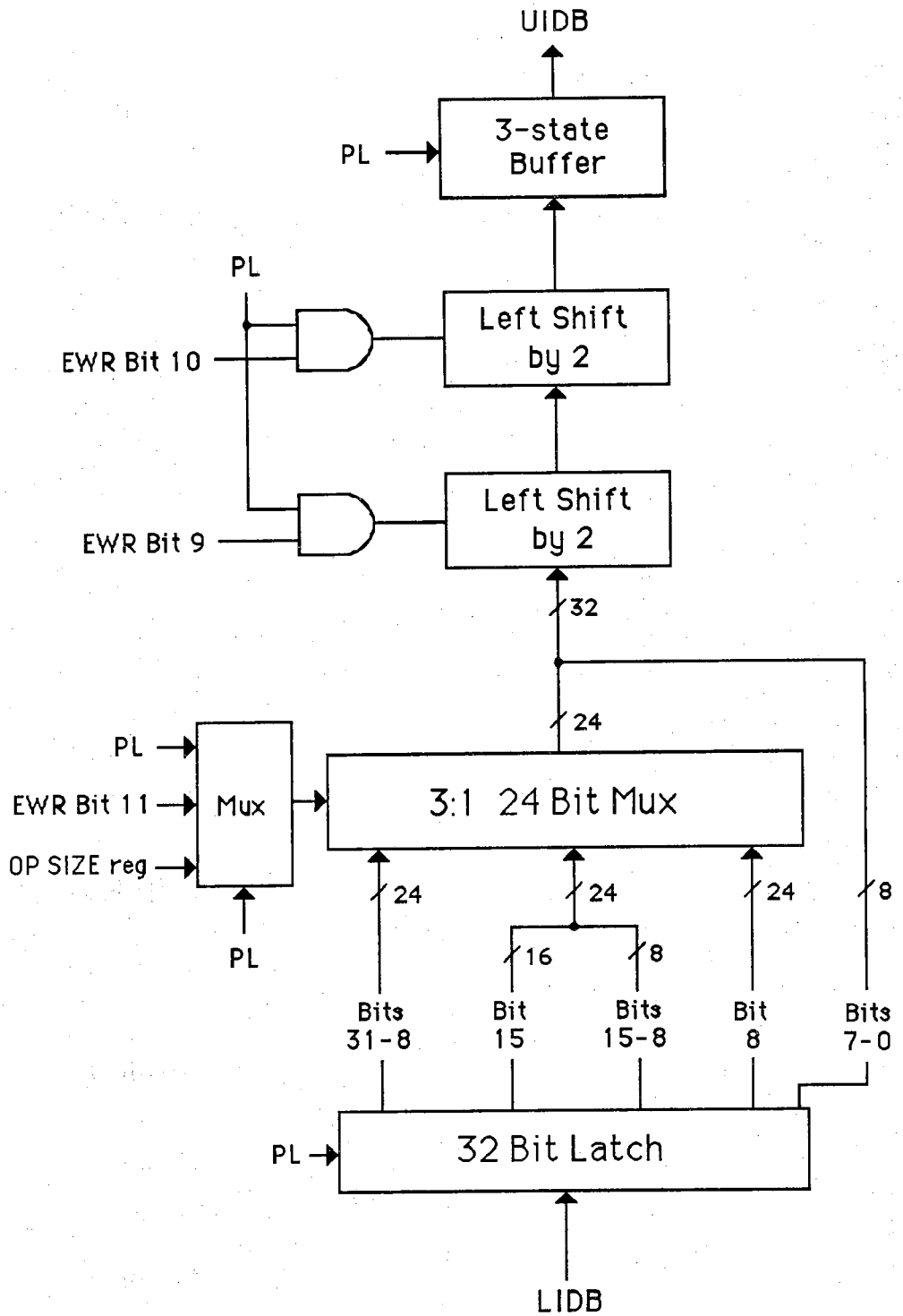
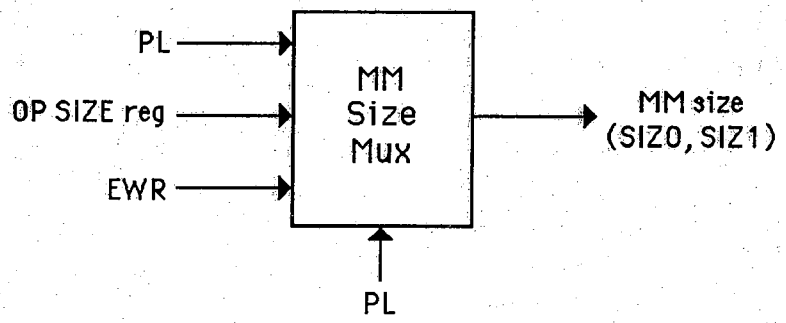

UIDB
3-state Buffer
PL
PL
Left Shift by 2
EWR Bit 10
Left Shift by 2
EWR Bit 9
32
24
PL
EWR Bit 11
OP SIZE reg
Mux
PL
3:1 24 Bit Mux
24
24
24
8
16
8
Bits 31-8
Bit 15
Bits 15-8
Bit 8
Bits 7-0
PL
32 Bit Latch
LIDB

Figure 10


PL
OP SIZE reg
EWR
MM
Size
Mux
MM size
(SIZ0, SIZ1)
PL

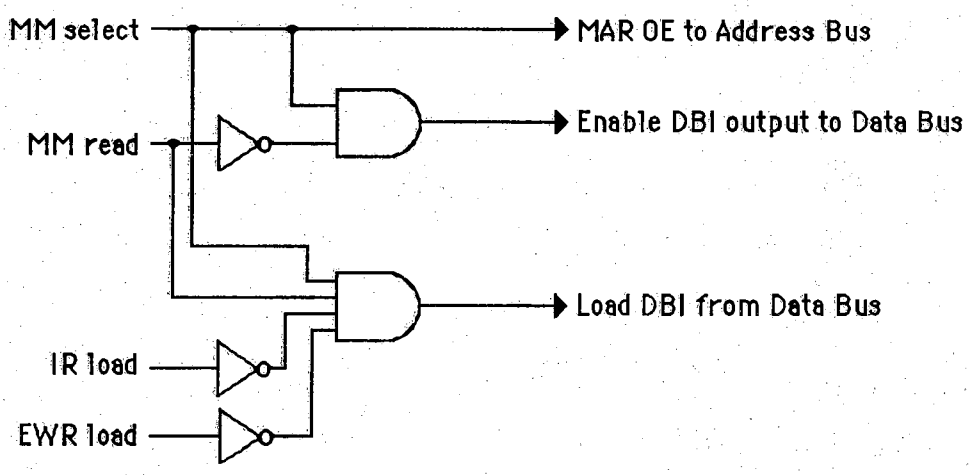

MM select
MM read
IR load
EWR load
MAR OE to Address Bus
Enable DBI output to Data Bus
Load DBI from Data Bus

Figure 11

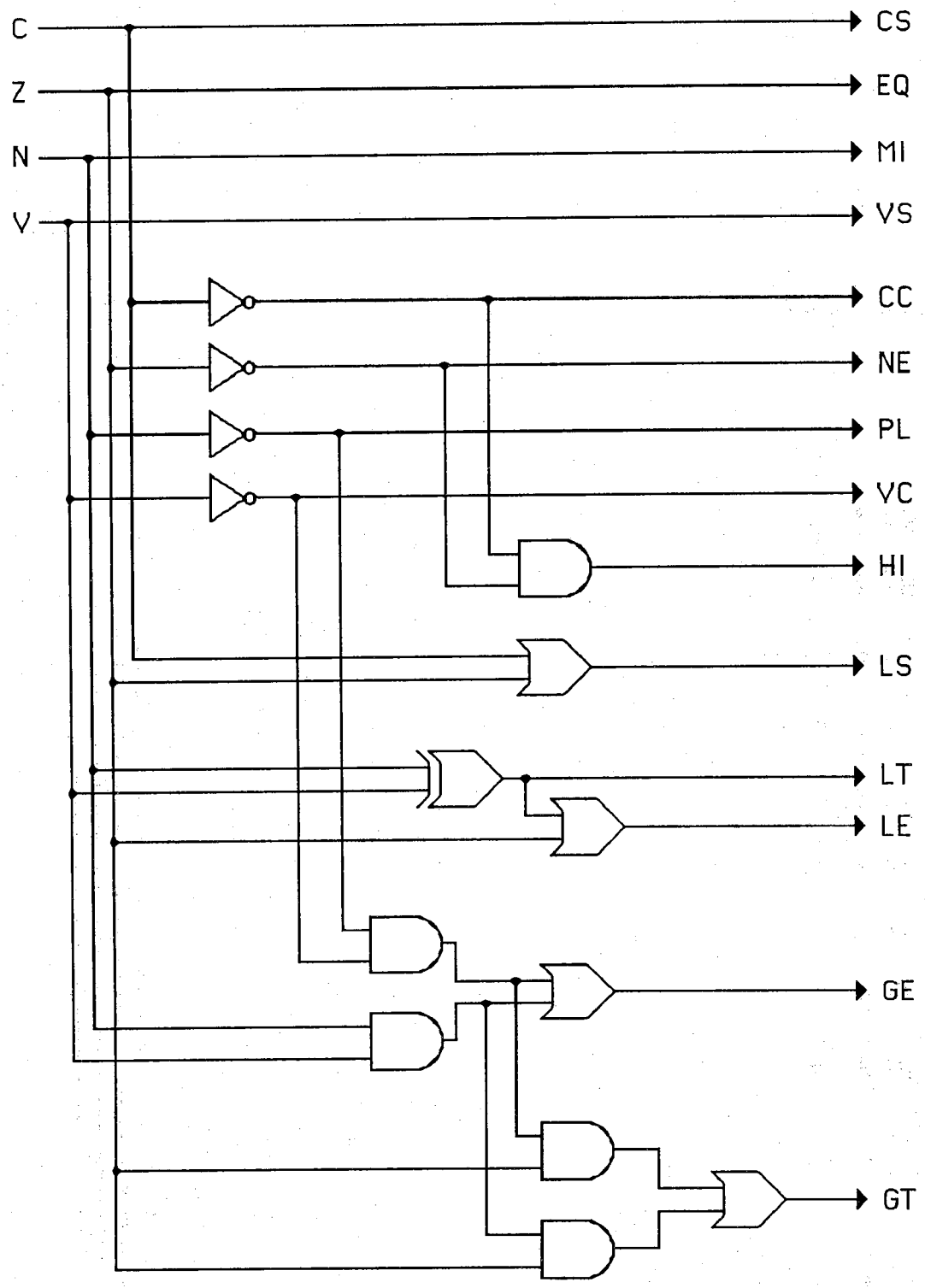


Figure 12

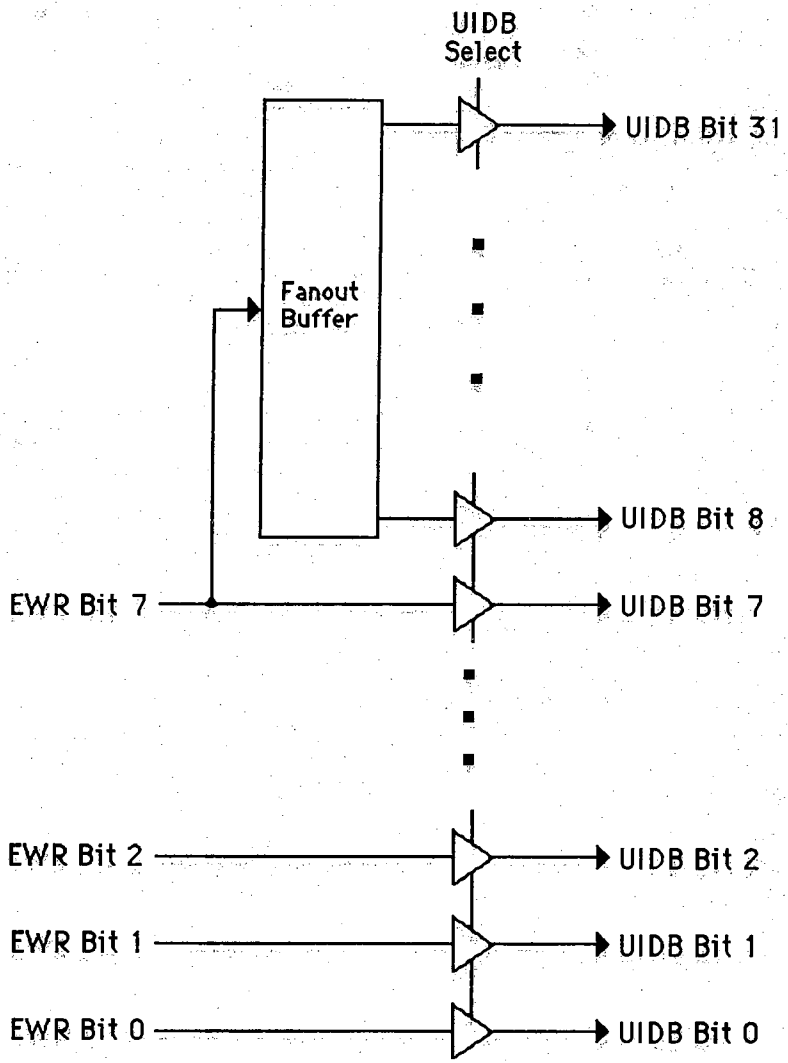

UIDB
Select
UIDB Bit 31
Fanout
Buffer
UIDB Bit 8
EWR Bit 7
UIDB Bit 7
EWR Bit 2
UIDB Bit 2
EWR Bit 1
UIDB Bit 1
EWR Bit 0
UIDB Bit 0

Figure 13

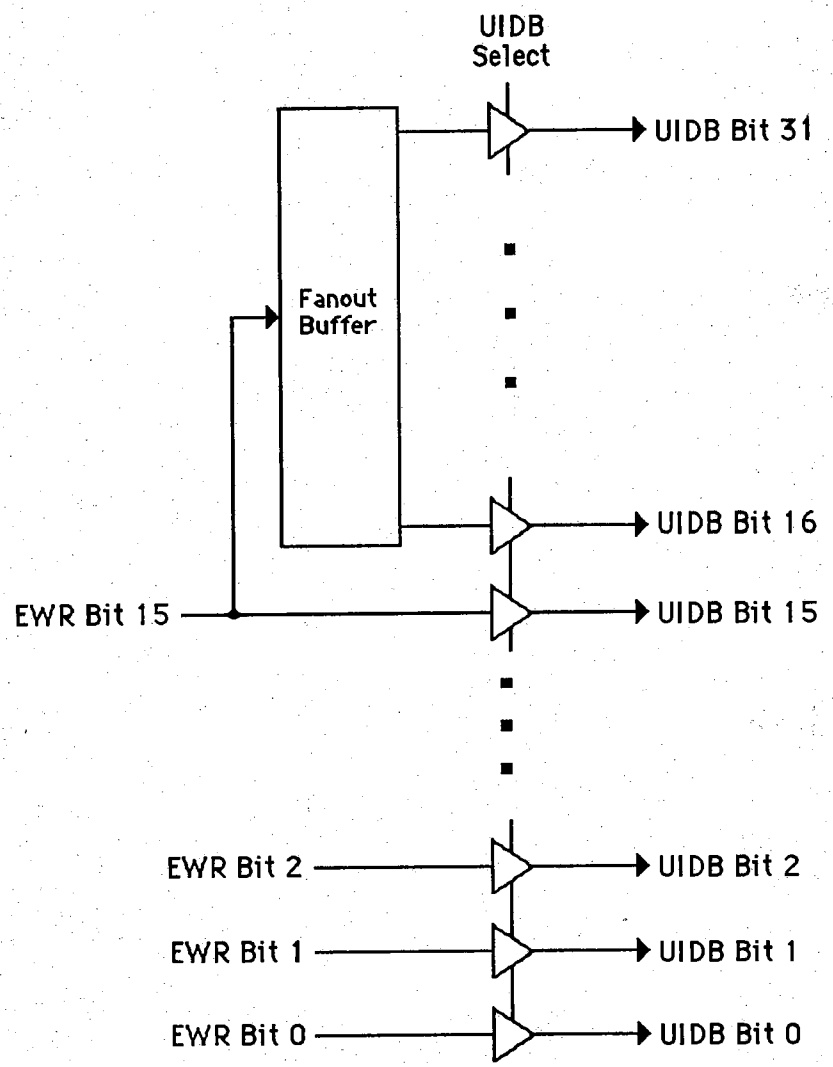

UIDB
Select
UIDB Bit 31
Fanout
Buffer
UIDB Bit 16
EWR Bit 15
UIDB Bit 15
EWR Bit 2
UIDB Bit 2
EWR Bit 1
UIDB Bit 1
EWR Bit 0
UIDB Bit 0

Figure 14

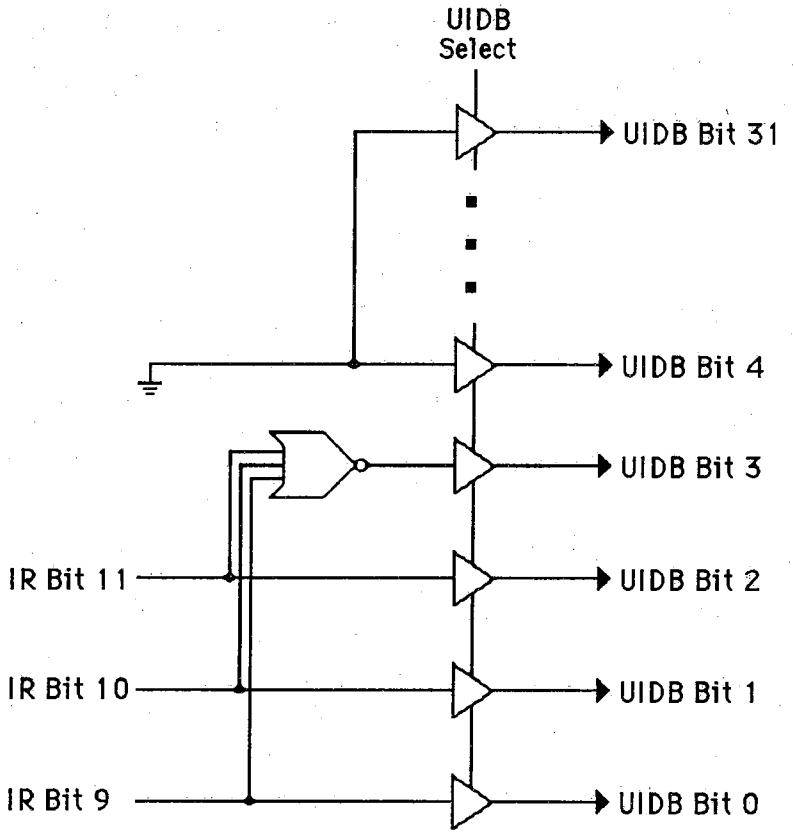

UIDB
Select
UIDB Bit 31
UIDB Bit 4
UIDB Bit 3
IR Bit 11
UIDB Bit 2
IR Bit 10
UIDB Bit 1
IR Bit 9
UIDB Bit 0

Figure 15

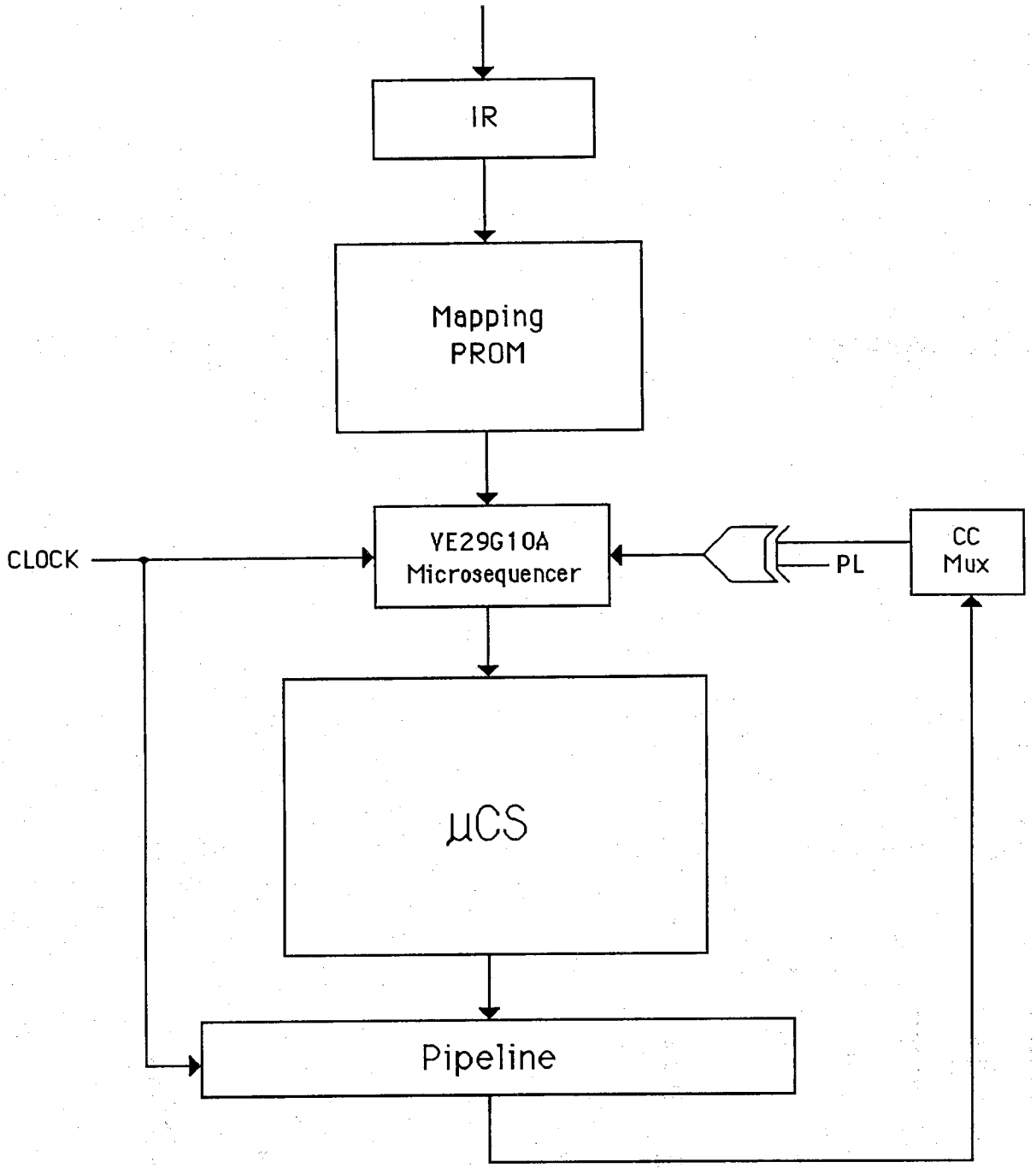

IR
Mapping
PROM
CLOCK
VE29G10A
Microsequencer
PL
CC
Mux
µCS
Pipeline

Figure 16

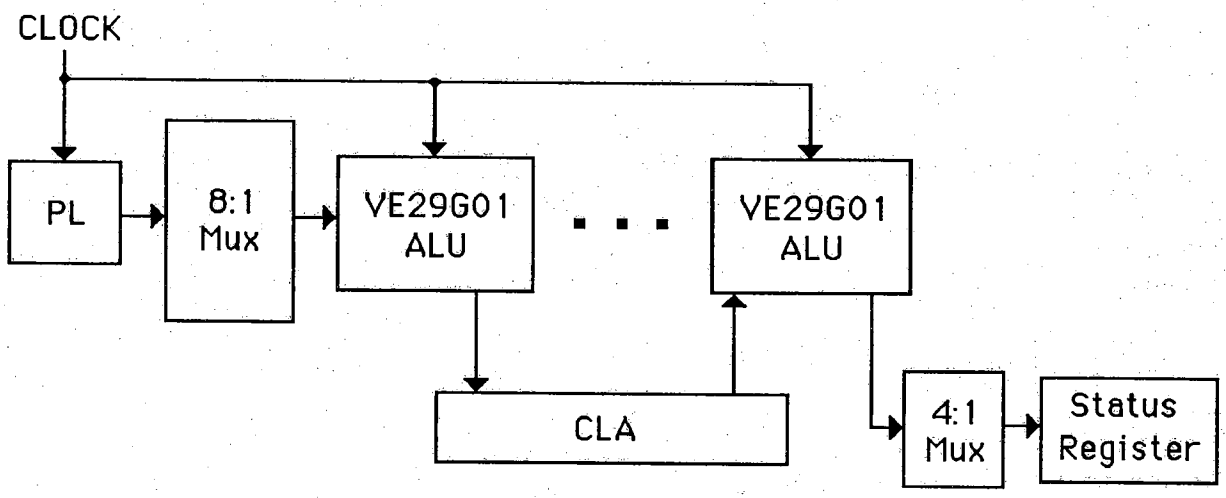

CLOCK
PL
8:1
Mux
VE29G01
ALU
VE29G01
ALU
CLA
4:1
Mux
Status
Register

Figure 17

Figure and Table Cross Reference List

# in this text	# in original text	Original Reference
Figure 1	Figure 1-1	1
Figure 2	Figure 1-2	1
Figure 3	None	2
Figure 4	None	2
Table 1	Figure 3	2
Table 2	Figure 4	2
Table 3	Figure 5	2

References

1. Motorola, Inc., *MC68020 32-bit Microprocessor User's Manual*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

2. Vitesse Semiconductor Corporation, VS29G01 and VS29G10A Data Sheets, 741 Calle Plano, Camarillo, California, 93010.

Appendix A

Microinstruction Fields

Appendix A
Microinstruction Fields

NAME	# of BITS	DEFINITION
useq.NA	12	Next address
useq.inst	4	Microsequencer instruction 0000 = JUMP ZERO 0001 = COND JSB PL 0010 = JUMP MAP 0011 = COND JUMP PL 0100 = PUSH/COND LD CNTR 0101 = COND JSB R/PL 0110 = COND JUMP VECTOR 0111 = COND JUMP R/PL 1000 = REPEAT LOOP, CNTR <> 0 1001 = REPEAT PL, CNTR <> 0 1010 = COND RTN 1011 = COND JUMP PL & POP 1100 = LD CNTR & CONTINUE 1101 = TEST END LOOP 1110 = CONTINUE 1111 = THREE-WAY BRANCH
useq.map_sel	2	Selects the 2910's mapping PROM 00 = PL Next Address 01 = EA PROM 10 = IR PROM
useq.CC_logic_level	1	Selects the true logic level for the CC to microsequencer 0 = active low 1 = active high

useq.CC_sel	7	Selects CC to microsequencer 000bbbb = EWR bit (bbbb) 001bbbb = IR bit (bbbb) 01bbbbb = LIDB bit (bbbbb) 100xxxx = Target level CC HI 100xxxx = Target level CC LS 100xxxx = Target level CC CC 100xxxx = Target level CC CS (C) 100xxxx = Target level CC NE 100xxxx = Target level CC EQ (Z) 100xxxx = Target level CC VC 100xxxx = Target level CC VS (V) 100xxxx = Target level CC PL 100xxxx = Target level CC MI (N) 100xxxx = Target level CC GE 100xxxx = Target level CC LT 100xxxx = Target level CC GT 100xxxx = Target level CC LE 100xxxx = Target level CC X 1010000 = Target level CC Z 1010011 = Logical True 1010100 = Logical False 1010101 = Host level CC Z 1010110 = Host level CC N 1010111 = Host level CC V 1011000 = Host level CC C Where xxxx = IR bits 11-8
IR_load	1	IR load line 0 = Hold IR 1 = Load IR with data bus (D15-D0)
EWR_load	1	EWR load line 0 = Hold EWR 1 = Load EWR with data bus (D15-D0)
op_size.load	1	Op_size load line 0 = Hold op_size 1 = Load op_size
op_size.select	2	Selects the op_size source 00 = PL 01 = IR bits 7-6 10 = IR bits 10-9 11 = IR bits 13-12
op_size.data	2	Sets the op_size register 00 = Byte 01 = Word 10 = Long word

UIDB_sel	4	UIDB device selector 0000 = No device selected 0001 = DBI 0010 = Constants PROM 0011 = op_size offset PROM 0100 = Target SR 0101 = EWR bits 7-0 (sign extended) 0110 = Feedback latch 0111 = Shifter 1000 = Lower multiplier result 1001 = Upper multiplier result 1010 = EWR bits 15-0 (sign extended) 1011 = IR bits 11-9 (0 --> 8) 1100 = IR bits 7-0 (sign extended)
data_ALU.dest	3	Data ALU instruction destination 000 = QREG 001 = NOP 010 = RAMA 011 = RAMF 100 = RAMQD 101 = RAMD 110 = RAMQU 111 = RAMU
data_ALU.funct	3	Data ALU instruction function 000 = ADD 001 = SUBR 010 = SUBS 011 = OR 100 = AND 101 = NOTRS 110 = EXOR 111 = EXNOR
data_ALU.src	3	Data ALU instruction source 000 = AQ 001 = AB 010 = ZQ 011 = ZB 100 = ZA 101 = DA 110 = DQ 111 = DZ
data_ALU.OE	1	Data ALU output enable 0 = Disable data ALU output 1 = Enable data ALU output
data_ALU.A_bus	4	A_bus for Data ALU

data_ALU.A_bus_sel	3	Selects the Data ALU A_bus source 000 = PL 001 = IR bits 2-0 010 = IR bits 8-6 011 = IR bits 11-9 100 = IR bits 14-12 101 = EWR bits 14-12 110 = EAR bits 2-0 111 = EWR bits 2-0 Note: the MSB of the A_bus is a hardwired 0 for sources with only 3 bits
data_ALU.B_bus	4	B_bus for Data ALU
data_ALU.B_bus_sel	3	Selects the Data ALU B_bus source 000 = PL 001 = IR bits 2-0 010 = IR bits 8-6 011 = IR bits 11-9 100 = IR bits 14-12 101 = EWR bits 14-12 110 = EAR bits 2-0 111 = EWR bits 2-0 Note: the MSB of the B_bus is a hardwired 0 for sources with only 3 bits
addr_ALU.dest	3	Address ALU instruction destination 000 = QREG 001 = NOP 010 = RAMA 011 = RAMF 100 = RAMQD 101 = RAMD 110 = RAMQU 111 = RAMU
addr_ALU.funct	3	Address ALU instruction function 000 = ADD 001 = SUBR 010 = SUBS 011 = OR 100 = AND 101 = NOTRS 110 = EXOR 111 = EXNOR

addr_ALU.src	3	Address ALU instruction source 000 = AQ 001 = AB 010 = ZQ 011 = ZB 100 = ZA 101 = DA 110 = DQ 111 = DZ
addr_ALU.OE	1	Address ALU output enable 0 = Disable address ALU output 1 = Enable address ALU output
addr_ALU.A_bus	4	A_bus for Address ALU
addr_ALU.A_bus_sel	3	Selects the Address ALU A_bus source 000 = PL 001 = IR bits 2-0 010 = IR bits 8-6 011 = IR bits 11-9 100 = IR bits 14-12 101 = EWR bits 14-12 110 = EAR bits 2-0 111 = EWR bits 2-0 Note: the MSB of the A_bus is a hardwired 0 for sources with only 3 bits
addr_ALU.B_bus	4	B_bus for Address ALU
addr_ALU.B_bus_sel	3	Selects the Address ALU B_bus source 000 = PL 001 = IR bits 2-0 010 = IR bits 8-6 011 = IR bits 11-9 100 = IR bits 14-12 101 = EWR bits 14-12 110 = EAR bits 2-0 111 = EWR bits 2-0 Note: the MSB of the B_bus is a hardwired 0 for sources with only 3 bits
MAR.load	1	MAR load line 0 = Hold MAR 1 = Load MAR with lower internal data bus (LIDB) value

const_addr	4	Selects constant from PROM 0000 = 00000002 H 0001 = 00000004 H 0010 = 00000008 H 0011 = 000000FF H 0100 = FFFFFF00 H 0101 = 0000FFFF H 0110 = FFFF0000 H 0111 = 00000001 H 1000 = 00000010 H 1001 = FFFF8000 H 1010 = 00010000 H
DBI.load	1	DBI load line 0 = Hold DBI 1 = Load DBI with UIDB
MM.select	1	Select line for main memory (MM) 0 = Do not select 1 = Select
MM.read	1	Read/Write line for MM 0 = Write 1 = Read
MM.size_sel	2	Select MM size source 00 = PL 01 = op_size register 10 = EWR bits 1-0 11 = EWR bits 5-4
MM.size	2	Sets MM size 00 = Byte 01 = Word 10 = Long word
TCC.X_load	1	Target CC X load line 0 = Hold target X flag 1 = Load target X flag
TCC.N_load	1	Target CC N load line 0 = Hold target N flag 1 = Load target N flag
TCC.Z_load	1	Target CC Z load line 0 = Hold target Z flag 1 = Load target Z flag
TCC.V_load	1	Target CC V load line 0 = Hold target V flag 1 = Load target V flag

TCC.C_load	1	Target CC C load line 0 = Hold target C flag 1 = Load target C flag
SR_sel	2	Selects the SR source 00 = Data ALU 01 = UIDB 10 = Shifter 11 = Address ALU
Cin.data	1	Carry in bit of Data ALU
Cin.select	1	Selects the Cin source 0 = PL 1 = Target level SR
mult.data_1_load	1	Loads multiplier latch #1 0 = Hold 1 = Load
mult.data_2_load	1	Loads multiplier latch #2 0 = Hold 1 = Load
shifter.inst	3	Shifter instruction (Hold two cycles) 000 = Arithmetic shift right 001 = Arithmetic shift left 010 = Logical shift right 011 = Logical shift left 100 = Rotate right with extend 101 = Rotate left with extend 110 = Rotate right without extend 111 = Rotate left without extend
shifter.shift_load	1	Loads barrel shifter shift register 0 = Hold 1 = Load
shifter.data_load	1	Loads barrel shifter data register 0 = Hold 1 = Load
shifter.CX_load	1	Loads C and X registers 0 = Hold 1 = Load
feedback.load	1	Loads feedback latch 0 = Hold 1 = Load

feedback.size_sel	2	Selects feedback sign extension size 00 = PL 01 = op_size 10 = EWR bit 11
feedback.size	2	Feedback sign extension size 00 = Byte 01 = Word 10 = Long word (disable)
feedback.scale_sel	1	Enables EWR scaling 0 = Disable scaling 1 = Enable scaling
EAR.load	1	EAR load line 0 = Hold EAR 1 = Load EAR
EAR.select	1	EAR / EA PROM select line 0 = IR bits 5-0 1 = IR bits 11-6
data_ALU_mask	1	Data ALU Mask Enable 0 = Disable Data ALU Masking 1 = Enable Data ALU Masking

Appendix B

Simulation Program

Appendix B
Simulation Program

```
/*****************************************************************************/
/*                                                                           */
/*   PROGRAM:  MAIN.C                                                        */
/*                                                                           */
/*   AUTHORS:  Erin Handgen and Bryan Robbins                                */
/*                                                                           */
/*   DATE:     March 2, 1988                                                 */
/*                                                                           */
/*   DESCRIPTION:                                                            */
/*        This program simulates a bit sliced, microprogrammed               */
/*        microprocessor that will emulate the Motorola 68020.               */
/*        The program is menu driven from a terminal and simulates           */
/*        micro-operations on the host.  The user is allowed                 */
/*        to change the state of the host, execute microinstructions         */
/*        from the microcontrol store, or execute machine instructions       */
/*        from the target level main memory.  After a microinstruction       */
/*        or machine instruction is executed, the new state of the           */
/*        host or target is reported to the user.                            */
/*                                                                           */
/*   INPUT:                                                                  */
/*        The program requires the microcontrol store bits to be             */
/*        stored in a file called 'uCS' and the target level main            */
/*        memory bits in a file called 'MM'.                                 */
/*                                                                           */
/*        uCS:                                                               */
/*             Each line in uCS is either a 123 bit microinstruction         */
/*             or a block address.  The 123 bit microinstruction line        */
/*             has 123 '1's or '0's.  A block address line begins with       */
/*             the letter 'a' and is followed by a 3 digit hex address       */
/*             that is used as the starting address for the block of         */
/*             microinstructions.                                            */
/*                                                                           */
/*        MM:                                                                */
/*             Each line in MM contains either a byte, word, longword        */
/*             or block address.  A byte, word, or longword line             */
/*             contains 8, 16, or 32 '1's or '0's respectively.  A           */
/*             block address line begins with the letter 'a' and is          */
/*             followed by a 4 digit hex address that is used as the         */
/*             starting address for the block of target level                */
/*             instructions.                                                 */
/*                                                                           */
/*   OUTPUT:                                                                 */
/*        All output is sent to the terminal.                                */
/*                                                                           */
/*****************************************************************************/
```

```
/* Include files */
#include <stdio.h>


/*****************************************************************************/
/* Defined constants                                                         */
/*****************************************************************************/

/* boolean true and false */
#define TRUE        1
#define FALSE       0

/* AM2901 destination constants */
#define QREG        0
#define NOP         1
#define RAMA        2
#define RAMF        3
#define RAMQD       4
#define RAMD        5
#define RAMQU       6
#define RAMU        7

/* AM2901 source constants */
#define AQ          0
#define AB          1
#define ZQ          2
#define ZB          3
#define ZA          4
#define DA          5
#define DQ          6
#define DZ          7

/* AM2901 function constants */
#define ADD         0
#define SUBR        1
#define SUBS        2
#define OR          3
#define AND         4
#define NOTRS       5
#define EXOR        6
#define EXNOR       7

/* AM2910 instructions */
#define JZ          0
#define CJS         1
#define JMAP        2
#define CJP         3
#define PUSH        4
#define JSRP        5
#define CJV         6
#define JRP         7
#define RFCT        8
#define RPCT        9
#define CRTN        10
#define CJPP        11
#define LDCT        12
#define LOOP        13
#define CONT        14
#define TWB         15
```

```
/*****************************************************************************/
/* Structure declarations                                                    */
/*****************************************************************************/

struct useq_PL_type {
     int NA;                          /* Next Address  */
     int inst;                        /* Microsequencer instruction */
     int map_sel;                     /* Mapping PROM select */
     int CC_logic_level;              /* Condition code logic level */
     int CC_sel;                      /* Condition code select */
};

struct op_size_PL_type {
     int load;                        /* Op size load */
     int select;                      /* Op size sel */
     int data;                        /* Op size data */
};

struct ALU_PL_type {
     int src;                         /* ALU instruction source */
     int funct;                       /* ALU instruction function */
     int dest;                        /* ALU instruction destination */
     int OE;                          /* ALU output enable */
     int A_bus;                       /* ALU A bus */
     int A_bus_sel;                   /* ALU A bus select */
     int B_bus;                       /* ALU B bus */
     int B_bus_sel;                   /* ALU B bus select */
};

struct MAR_PL_type {
     int load;                        /* MAR load */
};

struct DBI_PL_type {
     int load;                        /* Data bus interface load */
};

struct MM_PL_type {
     int select;                      /* Main memory select */
     int read;                        /* Main memory read */
     int size_sel;                    /* Main memory size select */
     int size;                        /* Main memory size */
};

struct TCC_PL_type {
     int X_load;                      /* X load */
     int N_load;                      /* N load */
     int Z_load;                      /* Z load */
     int V_load;                      /* V load */
     int C_load;                      /* C load */
};

struct Cin_PL_type {
     int data;                        /* Cin data */
     int select;                      /* Cin select */
};
```

```
struct mult_PL_type {
     int data_1_load;              /* Multiplier latch 1 load */
     int data_2_load;              /* Multiplier latch 2 load */
};

struct shifter_PL_type {
     int inst;                     /* Shifter instruction */
     int shift_load;               /* Shifter shift load line */
     int data_load;                /* Shifter data load */
     int CX_load;                  /* Shifter CX register load */
};

struct feedback_PL_type {
     int load;                     /* Feedback load */
     int size_sel;                 /* Feedback size select */
     int size;                     /* Feedback sign extension size */
     int scale_sel;                /* Feedback scale select */
};

struct EAR_PL_type {
     int load;                     /* EAR load */
     int select;                   /* EAR select line */
};

struct PL_type {
     struct useq_PL_type useq;          /* Microsequencer */
     int IR_load;                       /* Instruction register load */
     int EWR_load;                      /* Extended word register load */
     struct op_size_PL_type op_size;    /* Operand size */
     int UIDB_sel;                      /* UIDB device select */
     struct ALU_PL_type data_ALU;       /* Data ALU */
     struct ALU_PL_type addr_ALU;       /* Address ALU */
     struct MAR_PL_type MAR;            /* MAR */
     int const_addr;                    /* Constants PROM address */
     struct DBI_PL_type DBI;            /* DBI */
     struct MM_PL_type MM;              /* MM */
     struct TCC_PL_type TCC;            /* Target Condition Codes */
     int SR_sel;                        /* SR select */
     struct Cin_PL_type Cin;            /* Cin */
     struct mult_PL_type mult;          /* Multiplier */
     struct shifter_PL_type shifter;    /* Shifter */
     struct feedback_PL_type feedback; /* Feedback latch */
     struct EAR_PL_type EAR;            /* EAR */
     int data_ALU_mask;                 /* Data ALU mask enable */
};

struct ALU_state_type {
     int RAM[16];                  /* RAM registers */
     int Q;                        /* Q register */
};

struct CC_state_type {
     int X;                        /* Internal Carry */
     int C;                        /* Carry */
     int N;                        /* Negative */
     int V;                        /* Overflow */
     int Z;                        /* Zero */
};
```

```
struct stack_type {
     int top;                          /* Stack Top Index */
     int element[9];                   /* Stack Elements */
};

struct useq_state_type {
     int uPC;                          /* Microprogram counter */
     struct stack_type stack;          /* Microsequencer stack */
     int counter;                      /* Register/Counter */
     int CC;                           /* input CC */
};

struct mult_state_type {
     int data_1;                      /* Multiplier latch 1 */
     int data_2;                      /* Multiplier latch 2 */
};

struct shifter_state_type {
     int shift;                        /* Shift register */
     int data;                         /* Data register */
     int C;                            /* Shifter C register */
     int V;                            /* Shifter V register */
     int X;                            /* Shifter X register */
     int N;                            /* Shifter N register */
     int Z;                            /* Shifter Z register */
};

struct state_type {
     struct PL_type PL;                    /* Pipeline Register */
     struct ALU_state_type data_ALU;       /* Data ALU */
     struct ALU_state_type addr_ALU;       /* Address ALU */
     int DBI;                              /* Data Bus Interface */
     int MAR;                              /* Memory Address Register */
     struct CC_state_type TCC;             /* Target Condition Codes */
     struct CC_state_type HCC;             /* Host Condition Codes */
     int op_size;                          /* Operand Size */
     int PIR;                              /* Pre-Instruction Register */
     int IR;                               /* Instruction Register */
     int EWR;                              /* Extension Word Register */
     struct useq_state_type useq;          /* Microsequencer */
     struct mult_state_type mult;          /* Multiplier */
     struct shifter_state_type shifter;    /* Shifter */
     int feedback;                         /* Feedback Latch */
     int EAR;                              /* EAR */
     int LIDB;                             /* LIDB latch */
};

struct MM_cntl_type {
     int size;                        /* Data Size */
     int select;                      /* Main Memory Select */
     int read;                        /* Main Memory Read */
};
```

```
struct bus_type {
     int LIDB;                      /* Lower Internal Data Bus */
     int UIDB;                      /* Upper Internal Data Bus */
     int useq;                      /* microsequencer bus */
     int data_ALU_A;                /* Data ALU A Bus */
     int data_ALU_B;                /* Data ALU B Bus */
     int addr_ALU_A;                /* Address ALU A Bus */
     int addr_ALU_B;                /* Address ALU B Bus */
     struct MM_cntl_type MM_cntl;   /* Main Memory Control Bus */
     int MM_address;                /* Main Memory Address Bus */
     int MM_data;                   /* Main Memory Data Bus */
};


/*************************************************************************/
/* Global variables                                                      */
/*************************************************************************/

/*NOBASE*/
struct PL_type uCS[4096];      /* microcontrol store */
/*NOBASE*/
int MM[65536];                 /* target main memory */
struct state_type state;       /* state of host */
struct bus_type bus;           /* host bus values */
int const[16];                 /* constants PROM */
char buf[10];                  /* buffer for ungetting characters */
int bufp;                      /* pointer to top of character buffer */
int trace;                     /* flag which enables/disables tracing */
int breakpoint;                /* breakpoint used for debugging */
int inst_cnt;                  /* instruction cycle counter */
int prog_cnt;                  /* program cycle counter */
int error;                     /* error flag */
int hex_string[100];           /* string for returning results */
                               /* of procedure hex() */
struct CC_state_type data_CC; /* Data ALU condition codes */
struct CC_state_type addr_CC; /* Address ALU condition codes */
```

```
/*****************************************************************************/
/* main:                                                                     */
/*    See description above.                                                 */
/*****************************************************************************/

main()
{
     char option;                    /* menu option */

     /* initialize the state of the program and read in the uCS and MM */
     initialize();

     /* continue getting commands and executing them until exit */
     do {
          option = main_menu();
          switch (option) {
               case 'a':
                    change_state();
                    break;
               case 'b':
                    execute_1_uinst();
                    break;
               case 'c':
                    execute_N_uinst();
                    break;
               case 'd':
                    execute_1_inst();
                    break;
               case 'e':
                    execute_N_inst();
                    break;
               case 'f':
                    execute_prog();
                    break;
               case 'g':
                    print_host_state();
                    break;
               case 'h':
                    print_target_state();
                    break;
               case 'i':
                    edit_uCS();
                    break;
               case 'j':
                    edit_MM();
                    break;
               case 'k':
                    if (trace == 1)
                         trace = 0;
                    else
                         trace = 1;
                    break;
               case 'l':
                    breaks();
                    break;
```

```
                case 'm':
                     printf ("\n\nNew PC = ");
                     scanf ("%x", &state.addr_ALU.RAM[8]);
                     state.useq.uPC = 0;
                     break;
                case 'n':
                     printf("\n");
                     printf("Are you sure you want to quit? (Y/N)  ");
                     option = tolower(getchar());
                     while (getchar() != '\n')
                          ;
                     if (option == 'y')
                          option = 'o';
                     else
                          option = 'n';
                     break;
          };
     } while (option != 'o');

     /* save uCS */
     printf("\n");
     do {
          printf("Do you want to save the current uCS? (Y/N)  ");
          option = tolower(getchar());
          while (getchar() != '\n')
               ;
     } while ((option != 'y') && (option != 'n'));

     if (option == 'y')
          save_uCS();

     /* save MM */
     printf("\n");
     do {
          printf("Do you want to save the current MM? (Y/N)  ");
          option = tolower(getchar());
          while (getchar() != '\n')
               ;
     } while ((option != 'y') && (option != 'n'));

     if (option == 'y')
          save_MM();
}
```

```
/*****************************************************************************/
/* initialize:                                                               */
/*    Routine to initialize the global state variables and read in the       */
/*    microcontrol store and main memory.                                    */
/*****************************************************************************/

initialize()
{
     int index;                         /* array index */

     /* initialize breakpoint to 4096 */
     breakpoint = 4096;

     /* initialize cycle counters */
     prog_cnt = 0;
     inst_cnt = 0;

     /* initialize trace mode to ON */
     trace = TRUE;

     /* initialize buffer pointer */
     bufp = 0;

     /* clear all registers in data and address ALU's */
     for (index = 0; index < 16; index++) {
          state.data_ALU.RAM[index] = 0;
          state.addr_ALU.RAM[index] = 0;
     };
     state.data_ALU.Q = 0;
     state.addr_ALU.Q = 0;

     /* clear various host registers */
     state.DBI = 0;
     state.MAR = 0;
     state.op_size = 0;
     state.PIR = 0;
     state.IR = 0;
     state.EWR = 0;
     state.feedback = 0;
     state.EAR = 0;
     state.LIDB = 0;

     /* clear condition codes */
     state.HCC.Z = 0;
     state.HCC.V = 0;
     state.HCC.N = 0;
     state.HCC.C = 0;
     state.TCC.X = 0;
     state.TCC.Z = 0;
     state.TCC.V = 0;
     state.TCC.N = 0;
     state.TCC.C = 0;

     /* clear uPC and the uPC stack */
     state.useq.uPC = 0;
     state.useq.stack.top = 0;
```

```
        /* clear multiplier */
        state.mult.data_1 = 0;
        state.mult.data_2 = 0;

        /* clear barrel shifter */
        state.shifter.shift = 0;
        state.shifter.data = 0;

        /* clear main memory data bus */
        bus.MM_data = 0;

        /* read in microcontrol store */
        read_uCS();

        /* read in main memory */
        read_MM();

        /* load values into constants array */
        const[0] = 0x00000002;
        const[1] = 0x00000004;
        const[2] = 0x00000008;
        const[3] = 0x000000ff;
        const[4] = 0xffffff00;
        const[5] = 0x0000ffff;
        const[6] = 0xffff0000;
        const[7] = 0x00000001;
        const[8] = 0x00000010;
        const[9] = 0xffff8000;
        const[10] = 0x00010000;
        const[11] = 0x00000000;
        const[12] = 0x00000000;
        const[13] = 0x00000000;
        const[14] = 0x00000000;
        const[15] = 0x00000000;
}
```

```
/*************************************************************************/
/* read_uCS:                                                             */
/*   Routine that reads the contents of the global uCS array from the    */
/*   file called 'uCS'.  The file should contain one microinstruction    */
/*   per line.  The code is sequentially put in the uCS array starting   */
/*   at the array index indicated by the address line in the uCS file.   */
/*   An address line is a line that begins with the letter 'a' and is    */
/*   is followed by the HEX value of the new index.  The address lines   */
/*   must be in ascending order in the file!                             */
/*************************************************************************/

read_uCS()
{
     FILE *fp;                          /* file pointer */
     FILE *fopen();                     /* file open function */
     int new_index;                     /* new index to current uCS element */
     int index;                         /* index to current uCS element */
     char c;                            /* character */

     fp = fopen("uCS", "r");
     index = 0;
     while ((c = getch(fp)) != EOF) {
          if (c == 'a') {
               fscanf (fp, "%x", &new_index);
               getch(fp);

               /* clear unused uCS */
               while (index < new_index) {
                    uCS[index].useq.NA = 0;
                    uCS[index].useq.inst = 0;
                    uCS[index].useq.map_sel = 0;
                    uCS[index].useq.CC_logic_level = 0;
                    uCS[index].useq.CC_sel = 0;
                    uCS[index].IR_load = 0;
                    uCS[index].EWR_load = 0;
                    uCS[index].op_size.load = 0;
                    uCS[index].op_size.select = 0;
                    uCS[index].op_size.data = 0;
                    uCS[index].UIDB_sel = 0;
                    uCS[index].data_ALU.dest = 0;
                    uCS[index].data_ALU.funct = 0;
                    uCS[index].data_ALU.src = 0;
                    uCS[index].data_ALU.OE = 0;
                    uCS[index].data_ALU.A_bus = 0;
                    uCS[index].data_ALU.A_bus_sel = 0;
                    uCS[index].data_ALU.B_bus = 0;
                    uCS[index].data_ALU.B_bus_sel = 0;
                    uCS[index].addr_ALU.dest = 0;
                    uCS[index].addr_ALU.funct = 0;
                    uCS[index].addr_ALU.src = 0;
                    uCS[index].addr_ALU.OE = 0;
                    uCS[index].addr_ALU.A_bus = 0;
                    uCS[index].addr_ALU.A_bus_sel = 0;
                    uCS[index].addr_ALU.B_bus = 0;
                    uCS[index].addr_ALU.B_bus_sel = 0;
                    uCS[index].MAR.load = 0;
                    uCS[index].const_addr = 0;
                    uCS[index].DBI.load = 0;
                    uCS[index].MM.select = 0;
```

```
                uCS[index].MM.read = 0;
                uCS[index].MM.size_sel = 0;
                uCS[index].MM.size = 0;
                uCS[index].TCC.X_load = 0;
                uCS[index].TCC.N_load = 0;
                uCS[index].TCC.Z_load = 0;
                uCS[index].TCC.V_load = 0;
                uCS[index].TCC.C_load = 0;
                uCS[index].SR_sel = 0;
                uCS[index].Cin.data = 0;
                uCS[index].Cin.select = 0;
                uCS[index].mult.data_1_load = 0;
                uCS[index].mult.data_2_load = 0;
                uCS[index].shifter.inst = 0;
                uCS[index].shifter.shift_load = 0;
                uCS[index].shifter.data_load = 0;
                uCS[index].shifter.CX_load = 0;
                uCS[index].feedback.load = 0;
                uCS[index].feedback.size_sel = 0;
                uCS[index].feedback.size = 0;
                uCS[index].feedback.scale_sel = 0;
                uCS[index].EAR.load = 0;
                uCS[index].EAR.select = 0;
                uCS[index].data_ALU_mask = 0;
                index++;
        };
}
else {
        ungetch(c, fp);
        uCS[index].useq.NA = read_bits(12, fp);
        uCS[index].useq.inst = read_bits(4, fp);
        uCS[index].useq.map_sel = read_bits(2, fp);
        uCS[index].useq.CC_logic_level = read_bits(1, fp);
        uCS[index].useq.CC_sel = read_bits(7, fp);
        uCS[index].IR_load = read_bits(1, fp);
        uCS[index].EWR_load = read_bits(1, fp);
        uCS[index].op_size.load = read_bits(1, fp);
        uCS[index].op_size.select = read_bits(2, fp);
        uCS[index].op_size.data = read_bits(2, fp);
        uCS[index].UIDB_sel = read_bits(4, fp);
        uCS[index].data_ALU.dest = read_bits(3, fp);
        uCS[index].data_ALU.funct = read_bits(3, fp);
        uCS[index].data_ALU.src = read_bits(3, fp);
        uCS[index].data_ALU.OE = read_bits(1, fp);
        uCS[index].data_ALU.A_bus = read_bits(4, fp);
        uCS[index].data_ALU.A_bus_sel = read_bits(3, fp);
        uCS[index].data_ALU.B_bus = read_bits(4, fp);
        uCS[index].data_ALU.B_bus_sel = read_bits(3, fp);
        uCS[index].addr_ALU.dest = read_bits(3, fp);
        uCS[index].addr_ALU.funct = read_bits(3, fp);
        uCS[index].addr_ALU.src = read_bits(3, fp);
        uCS[index].addr_ALU.OE = read_bits(1, fp);
        uCS[index].addr_ALU.A_bus = read_bits(4, fp);
        uCS[index].addr_ALU.A_bus_sel = read_bits(3, fp);
        uCS[index].addr_ALU.B_bus = read_bits(4, fp);
        uCS[index].addr_ALU.B_bus_sel = read_bits(3, fp);
        uCS[index].MAR.load = read_bits(1, fp);
        uCS[index].const_addr = read_bits(4, fp);
        uCS[index].DBI.load = read_bits(1, fp);
```

```
            uCS[index].MM.select = read_bits(1, fp);
            uCS[index].MM.read = read_bits(1, fp);
            uCS[index].MM.size_sel = read_bits(2, fp);
            uCS[index].MM.size = read_bits(2, fp);
            uCS[index].TCC.X_load = read_bits(1, fp);
            uCS[index].TCC.N_load = read_bits(1, fp);
            uCS[index].TCC.Z_load = read_bits(1, fp);
            uCS[index].TCC.V_load = read_bits(1, fp);
            uCS[index].TCC.C_load = read_bits(1, fp);
            uCS[index].SR_sel = read_bits(2, fp);
            uCS[index].Cin.data = read_bits(1, fp);
            uCS[index].Cin.select = read_bits(1, fp);
            uCS[index].mult.data_1_load = read_bits(1, fp);
            uCS[index].mult.data_2_load = read_bits(1, fp);
            uCS[index].shifter.inst = read_bits(3, fp);
            uCS[index].shifter.shift_load = read_bits(1, fp);
            uCS[index].shifter.data_load = read_bits(1, fp);
            uCS[index].shifter.CX_load = read_bits(1, fp);
            uCS[index].feedback.load = read_bits(1, fp);
            uCS[index].feedback.size_sel = read_bits(2, fp);
            uCS[index].feedback.size = read_bits(2, fp);
            uCS[index].feedback.scale_sel = read_bits(1, fp);
            uCS[index].EAR.load = read_bits(1, fp);
            uCS[index].EAR.select = read_bits(1, fp);
            uCS[index].data_ALU_mask = read_bits(1, fp);
            index++;
            getch(fp);
        }
};

/* clear unused uCS */
while (index < 4096) {
        uCS[index].useq.NA = 0;
        uCS[index].useq.inst = 0;
        uCS[index].useq.map_sel = 0;
        uCS[index].useq.CC_logic_level = 0;
        uCS[index].useq.CC_sel = 0;
        uCS[index].IR_load = 0;
        uCS[index].EWR_load = 0;
        uCS[index].op_size.load = 0;
        uCS[index].op_size.select = 0;
        uCS[index].op_size.data = 0;
        uCS[index].UIDB_sel = 0;
        uCS[index].data_ALU.dest = 0;
        uCS[index].data_ALU.funct = 0;
        uCS[index].data_ALU.src = 0;
        uCS[index].data_ALU.OE = 0;
        uCS[index].data_ALU.A_bus = 0;
        uCS[index].data_ALU.A_bus_sel = 0;
        uCS[index].data_ALU.B_bus = 0;
        uCS[index].data_ALU.B_bus_sel = 0;
        uCS[index].addr_ALU.dest = 0;
        uCS[index].addr_ALU.funct = 0;
        uCS[index].addr_ALU.src = 0;
        uCS[index].addr_ALU.OE = 0;
        uCS[index].addr_ALU.A_bus = 0;
        uCS[index].addr_ALU.A_bus_sel = 0;
        uCS[index].addr_ALU.B_bus = 0;
        uCS[index].addr_ALU.B_bus_sel = 0;
```

```
            uCS[index].MAR.load = 0;
            uCS[index].const_addr = 0;
            uCS[index].DBI.load = 0;
            uCS[index].MM.select = 0;
            uCS[index].MM.read = 0;
            uCS[index].MM.size_sel = 0;
            uCS[index].MM.size = 0;
            uCS[index].TCC.X_load = 0;
            uCS[index].TCC.N_load = 0;
            uCS[index].TCC.Z_load = 0;
            uCS[index].TCC.V_load = 0;
            uCS[index].TCC.C_load = 0;
            uCS[index].SR_sel = 0;
            uCS[index].Cin.data = 0;
            uCS[index].Cin.select = 0;
            uCS[index].mult.data_1_load = 0;
            uCS[index].mult.data_2_load = 0;
            uCS[index].shifter.inst = 0;
            uCS[index].shifter.shift_load = 0;
            uCS[index].shifter.data_load = 0;
            uCS[index].shifter.CX_load = 0;
            uCS[index].feedback.load = 0;
            uCS[index].feedback.size_sel = 0;
            uCS[index].feedback.size = 0;
            uCS[index].feedback.scale_sel = 0;
            uCS[index].EAR.load = 0;
            uCS[index].EAR.select = 0;
            uCS[index].data_ALU_mask = 0;
            index++;
        };
        fclose(fp);
}
```

```
/*************************************************************************/
/* save_uCS:                                                             */
/*      Routine to save the current uCS array into a file called         */
/*      'uCS'.  Each uCS item is converted into a corresponding bit      */
/*      pattern and then sent to the file.  When an array element has    */
/*      all zero values in it, the routine will skip the block of zero   */
/*      elements and try to find the next used block of uCS.  When it    */
/*      is found, the address of the uCS is put in the file and the      */
/*      new block is written.                                            */
/*************************************************************************/

save_uCS()
{
     FILE *fp;                          /* file pointer */
     FILE *fopen();                     /* file open function */
     int new_index;                     /* new index to current uCS element */
     int index;                         /* index to current uCS element */
     char c;                            /* character */

     /* open the file */
     fp = fopen("uCS", "w");
     index = 0;

     /* write the uCS array out to the file */
     while (index < 4096) {

          /* skip empty uCS elements */
          while ((index < 4096) && (empty(uCS[index]) == TRUE))
               index++;

          /* print the address of the next block of useful uCS */
          if (index < 4096)
               fprintf(fp, "a%x\n", index);

          /* Write the block of nonzero uCS to file */
          while ((index < 4096) && (empty(uCS[index]) == FALSE)) {
               write_bits(uCS[index].useq.NA, 12, fp);
               write_bits(uCS[index].useq.inst, 4, fp);
               write_bits(uCS[index].useq.map_sel, 2, fp);
               write_bits(uCS[index].useq.CC_logic_level, 1, fp);
               write_bits(uCS[index].useq.CC_sel, 7, fp);
               write_bits(uCS[index].IR_load, 1, fp);
               write_bits(uCS[index].EWR_load, 1, fp);
               write_bits(uCS[index].op_size.load, 1, fp);
               write_bits(uCS[index].op_size.select, 2, fp);
               write_bits(uCS[index].op_size.data, 2, fp);
               write_bits(uCS[index].UIDB_sel, 4, fp);
               write_bits(uCS[index].data_ALU.dest, 3, fp);
               write_bits(uCS[index].data_ALU.funct, 3, fp);
               write_bits(uCS[index].data_ALU.src, 3, fp);
               write_bits(uCS[index].data_ALU.OE, 1, fp);
               write_bits(uCS[index].data_ALU.A_bus, 4, fp);
               write_bits(uCS[index].data_ALU.A_bus_sel, 3, fp);
               write_bits(uCS[index].data_ALU.B_bus, 4, fp);
               write_bits(uCS[index].data_ALU.B_bus_sel, 3, fp);
               write_bits(uCS[index].addr_ALU.dest, 3, fp);
               write_bits(uCS[index].addr_ALU.funct, 3, fp);
               write_bits(uCS[index].addr_ALU.src, 3, fp);
               write_bits(uCS[index].addr_ALU.OE, 1, fp);
```

```
                write_bits(uCS[index].addr_ALU.A_bus, 4, fp);
                write_bits(uCS[index].addr_ALU.A_bus_sel, 3, fp);
                write_bits(uCS[index].addr_ALU.B_bus, 4, fp);
                write_bits(uCS[index].addr_ALU.B_bus_sel, 3, fp);
                write_bits(uCS[index].MAR.load, 1, fp);
                write_bits(uCS[index].const_addr, 4, fp);
                write_bits(uCS[index].DBI.load, 1, fp);
                write_bits(uCS[index].MM.select, 1, fp);
                write_bits(uCS[index].MM.read, 1, fp);
                write_bits(uCS[index].MM.size_sel, 2, fp);
                write_bits(uCS[index].MM.size, 2, fp);
                write_bits(uCS[index].TCC.X_load, 1, fp);
                write_bits(uCS[index].TCC.N_load, 1, fp);
                write_bits(uCS[index].TCC.Z_load, 1, fp);
                write_bits(uCS[index].TCC.V_load, 1, fp);
                write_bits(uCS[index].TCC.C_load, 1, fp);
                write_bits(uCS[index].SR_sel, 2, fp);
                write_bits(uCS[index].Cin.data, 1, fp);
                write_bits(uCS[index].Cin.select, 1, fp);
                write_bits(uCS[index].mult.data_1_load, 1, fp);
                write_bits(uCS[index].mult.data_2_load, 1, fp);
                write_bits(uCS[index].shifter.inst, 3, fp);
                write_bits(uCS[index].shifter.shift_load, 1, fp);
                write_bits(uCS[index].shifter.data_load, 1, fp);
                write_bits(uCS[index].shifter.CX_load, 1, fp);
                write_bits(uCS[index].feedback.load, 1, fp);
                write_bits(uCS[index].feedback.size_sel, 2, fp);
                write_bits(uCS[index].feedback.size, 2, fp);
                write_bits(uCS[index].feedback.scale_sel, 1, fp);
                write_bits(uCS[index].EAR.load, 1, fp);
                write_bits(uCS[index].EAR.select, 1, fp);
                write_bits(uCS[index].data_ALU_mask, 1, fp);
                index++;
                fprintf(fp, "\n");
            };
        };
        fclose(fp);
}
```

```
/*************************************************************************/
/* read_MM:                                                              */
/*   Routine that reads the contents of the global MM array from the     */
/*   file called 'MM'.  The file should contain one byte                 */
/*   per line.  The code is sequentially put in the MM array starting    */
/*   at the array index indicated by the address line in the MM file.    */
/*   An address line is a line that begins with the letter 'a' and is    */
/*   is followed by the HEX value of the new index.  The address lines   */
/*   must be in ascending order in the file!                             */
/*************************************************************************/

read_MM()
{
     FILE *fp;                          /* file pointer */
     FILE *fopen();                     /* file open function */
     int new_index;                     /* new index to current MM element */
     int index;                         /* index to current MM element */
     char c;                            /* character */

     /* open MM file */
     fp = fopen("MM", "r");
     index = 0;

     /* read in MM */
     while ((c = getch(fp)) != EOF) {

          /* look for address line */
          if (c == 'a') {
               fscanf(fp, "%x", &new_index);
               getch(fp);

               /* clear unused MM */
               while (index < new_index) {
                    MM[index] = 0;
                    index++;
               }
          }

          /* else read in the next MM byte */
          else {
               ungetch(c, fp);
               MM[index] = read_bits(8, fp);
               index++;
               if ((c = getch(fp)) != '\n')
                    ungetch(c, fp);
          };
     };

     /* clear unused MM */
     while (index < 65536) {
          MM[index] = 0;
          index++;
     };
     fclose(fp);
}
```

```
/*************************************************************************/
/* save_MM:                                                              */
/*      Routine to save the current MM array into a file called          */
/*      'MM'.  Each MM item is converted into a corresponding bit        */
/*      pattern and then sent to the file.  When an array element has    */
/*      all zero values in it, the routine will skip the block of zero   */
/*      elements and try to find the next used block of MM.  When it     */
/*      is found, the address of the MM is put in the file and the       */
/*      new block is written.                                            */
/*************************************************************************/

save_MM()
{
     FILE *fp;                          /* file pointer */
     FILE *fopen();                     /* file open function */
     int new_index;                     /* new index to current MM element */
     int index;                         /* index to current MM element */
     char c;                            /* character */

     /* open MM file */
     fp = fopen("MM", "w");
     index = 0;

     /* look through the MM array to save it in the MM file */
     while (index < 65536) {

          /* skip zero MM elements */
          while ((index < 65536) && (MM[index] == 0))
               index++;

          /* put new address line in file */
          if (index < 65536)
               fprintf(fp, "a%x\n", index);

          /* write block of nonzero elements to file */
          while ((index < 65536) && (MM[index] != 0)) {
               write_bits(MM[index], 8, fp);
               index++;
               fprintf(fp, "\n");
          };
     };
     fclose(fp);
}
```

```
/*****************************************************************/
/* empty:                                                        */
/*     Routine that returns TRUE or FALSE depending on if all fields   */
/*     of the microinstruction are 0 or not.                     */
/*****************************************************************/

empty(uinst)
struct PL_type uinst;              /* microinstruction */
{
     int zero;                     /* instruction accumulator */

     zero = 0;
     zero = zero | uinst.useq.NA;
     zero = zero | uinst.useq.inst;
     zero = zero | uinst.useq.map_sel;
     zero = zero | uinst.useq.CC_logic_level;
     zero = zero | uinst.useq.CC_sel;
     zero = zero | uinst.IR_load;
     zero = zero | uinst.EWR_load;
     zero = zero | uinst.op_size.load;
     zero = zero | uinst.op_size.select;
     zero = zero | uinst.op_size.data;
     zero = zero | uinst.UIDB_sel;
     zero = zero | uinst.data_ALU.dest;
     zero = zero | uinst.data_ALU.funct;
     zero = zero | uinst.data_ALU.src;
     zero = zero | uinst.data_ALU.OE;
     zero = zero | uinst.data_ALU.A_bus;
     zero = zero | uinst.data_ALU.A_bus_sel;
     zero = zero | uinst.data_ALU.B_bus;
     zero = zero | uinst.data_ALU.B_bus_sel;
     zero = zero | uinst.addr_ALU.dest;
     zero = zero | uinst.addr_ALU.funct;
     zero = zero | uinst.addr_ALU.src;
     zero = zero | uinst.addr_ALU.OE;
     zero = zero | uinst.addr_ALU.A_bus;
     zero = zero | uinst.addr_ALU.A_bus_sel;
     zero = zero | uinst.addr_ALU.B_bus;
     zero = zero | uinst.addr_ALU.B_bus_sel;
     zero = zero | uinst.MAR.load;
     zero = zero | uinst.const_addr;
     zero = zero | uinst.DBI.load;
     zero = zero | uinst.MM.select;
     zero = zero | uinst.MM.read;
     zero = zero | uinst.MM.size_sel;
     zero = zero | uinst.MM.size;
     zero = zero | uinst.TCC.X_load;
     zero = zero | uinst.TCC.N_load;
     zero = zero | uinst.TCC.Z_load;
     zero = zero | uinst.TCC.V_load;
     zero = zero | uinst.TCC.C_load;
     zero = zero | uinst.SR_sel;
     zero = zero | uinst.Cin.data;
     zero = zero | uinst.Cin.select;
     zero = zero | uinst.mult.data_1_load;
     zero = zero | uinst.mult.data_2_load;
     zero = zero | uinst.shifter.inst;
     zero = zero | uinst.shifter.shift_load;
```

```
        zero = zero | uinst.shifter.data_load;
        zero = zero | uinst.shifter.CX_load;
        zero = zero | uinst.feedback.load;
        zero = zero | uinst.feedback.size_sel;
        zero = zero | uinst.feedback.size;
        zero = zero | uinst.feedback.scale_sel;
        zero = zero | uinst.EAR.load;
        zero = zero | uinst.EAR.select;
        zero = zero | uinst.data_ALU_mask;

        if (zero == 0)
            return(TRUE);
        else
            return(FALSE);
}


/*****************************************************************/
/* read_bits:                                                    */
/*   Function that reads the number of bits specified by width from  */
/*   the file pointed to by fp.  The bit field is then converted into */
/*   an integer and returned.                                    */
/*****************************************************************/

read_bits(width, fp)
int width;                          /* width of bit field */
FILE *fp;                           /* file pointer */
{
        int value;                  /* value of bit field */

        value = 0;
        while (width > 0) {
            value = value << 1;
            if (getch(fp) == '1')
                value++;
            --width;
        };
        return (value);
}
```

```
/*************************************************************************/
/* write_bits:                                                           */
/*      Routine that converts the integer value into a string of bits    */
/*      with length equal to width.  The string is then sent to the      */
/*      file pointed to by fp.                                           */
/*************************************************************************/

write_bits(value, width, fp)
int value;                      /* value to be converted */
int width;                      /* width of bit field */
FILE *fp;                       /* file pointer */
{
     int mask;                  /* bit mask */

     mask = 1 << width-1;
     while (width > 0) {
          if ((value & mask) == 0)
               fprintf(fp, "0");
          else
               fprintf(fp, "1");
          mask = mask >> 1;
          --width;
     };
}


/*************************************************************************/
/* getch:                                                                */
/*   Routine to get a character from file pointed to by fp.  This        */
/*   function works with ungetch() to simulate the ability to get        */
/*   and unget characters.                                               */
/*************************************************************************/

getch(fp)
FILE *fp;                       /* file pointer */
{
     return((bufp > 0) ? buf[--bufp] : getc(fp));
}


/*************************************************************************/
/* ungetch:                                                              */
/*   Routine to simulate putting a character back into the file          */
/*   pointed to by fp.                                                   */
/*************************************************************************/

ungetch(c, fp)
char c;                         /* character to be ungotten */
FILE *fp;                       /* file pointer */
{
     buf[bufp++] = c;
}
```

```
/*************************************************************************/
/* breaks:                                                               */
/*      Routine that sets or clears the breakpoint in the uCS array.     */
/*************************************************************************/

breaks()
{
     char option;

     option = breaks_menu();
     switch(option) {
          case 'a':
               printf("New breakpoint = ");
               scanf("%x", &breakpoint);
               while(getchar() != '\n')
                    ;
               break;
          case 'b':
               breakpoint = 4096;
               break;
     };
}


/*************************************************************************/
/* tolower:                                                              */
/*      Routine that converts a character to its lower case equivalent   */
/*************************************************************************/

tolower(ch)
char ch;                          /* character to be converted */
{
     if (('A' <= ch) && (ch <= 'Z'))
          return(ch - 'A' + 'a');
     else
          return(ch);
}


/*************************************************************************/
/* INCLUDE CHANGE                                                        */
/*************************************************************************/

#include "change.c"


/*************************************************************************/
/* INCLUDE MENUS                                                         */
/*************************************************************************/

#include "menu.c"


/*************************************************************************/
/* INCLUDE EXECUTE                                                       */
/*************************************************************************/

#include "execute.c"
```

```
/*****************************************************************************/
/* menu.c                                                                    */
/*                                                                           */
/*      Menu.c is a file that contains all of the routines that print        */
/*      menus on the terminal and waits for the user to enter an             */
/*      option.  Due to the simplicity of the code for the menu              */
/*      routines, functions will only be given a header with no              */
/*      in-depth explanations.  (How do you give an in-depth                 */
/*      explanation of a menu anyway?)                                       */
/*****************************************************************************/
```

```
/*************************************************************************/
/* main_menu                                                             */
/*************************************************************************/

main_menu()
{
     printf("\n\n");
     printf("=============================================================\n\n");
     printf("Main Menu\n\n");
     printf("   A ==> Change the state of the host\n");
     printf("   B ==> Execute 1 microinstruction\n");
     printf("   C ==> Execute N microinstructions\n");
     printf("   D ==> Execute 1 target instruction\n");
     printf("   E ==> Execute N target instructions\n");
     printf("   F ==> Execute target program\n");
     printf("   G ==> Print host state\n");
     printf("   H ==> Print target state\n");
     printf("   I ==> Edit microcontrol store\n");
     printf("   J ==> Edit main memory\n");
     if (trace == 1)
          printf("   K ==> Toggle trace mode  (currently ON)\n");
     else
          printf("   K ==> Toggle trace mode  (currently OFF)\n");
     printf("   L ==> Set breakpoint\n");
     printf("   M ==> Set PC\n");
     printf("   N ==> Quit program\n\n");
     printf("Enter option: ");
     return(get_option('n'));
}
```

```
/*****************************************************************/
/* change_state_menu:                                            */
/*****************************************************************/

change_state_menu()
{
     printf("\n\n");
     printf("=========================================================\n\n");
     printf("Change State Menu\n\n");
     printf("   A ==> Data ALU\n");
     printf("   B ==> Address ALU\n");
     hex(state.DBI, 8);
     printf("   C ==> Data Bus Interface (DBI)          %s\n", hex_string);
     hex(state.MAR, 8);
     printf("   D ==> Memory Address Register (MAR)     %s\n", hex_string);
     printf("   E ==> Target Condition Codes (TCC)\n");
     printf("   F ==> Operand Size Register (op_size)\n");
     hex(state.PIR, 4);
     printf("   G ==> Pre-Instruction Register (IR)        %s\n",
           hex_string);
     hex(state.IR, 4);
     printf("   H ==> Instruction Register (IR)            %s\n",
           hex_string);
     hex(state.EWR, 4);
     printf("   I ==> Extension Word Register (EWR)        %s\n",
           hex_string);
     printf("   J ==> Multiplier \n");
     printf("   K ==> Shifter \n");
     hex(state.feedback, 8);
     printf("   L ==> Feedback Register                 %s\n",
           hex_string);
     hex(state.EAR, 2);
     printf("   M ==> Effective Address Register (EAR)      %s\n",
           hex_string);
     hex(state.LIDB, 8);
     printf("   N ==> LIDB Register                     %s\n",
           hex_string);
     printf("   O ==> Microsequencer\n");
     printf("   P ==> Current Instruction in uCS\n");
     printf("   Q ==> Exit\n\n");
     printf("Enter option: ");
     return(get_option('q'));
}
```

```
/*************************************************************************/
/* data_ALU_state_menu:                                                  */
/*************************************************************************/

data_ALU_state_menu()
{
     printf("\n\n");
     printf("=========================================================\n\n");
     printf("Data ALU Menu\n\n");
     hex(state.data_ALU.RAM[0], 8);
     printf("   A ==> RAM  0   (D0)         %s\n", hex_string);
     hex(state.data_ALU.RAM[1], 8);
     printf("   B ==> RAM  1   (D1)         %s\n", hex_string);
     hex(state.data_ALU.RAM[2], 8);
     printf("   C ==> RAM  2   (D2)         %s\n", hex_string);
     hex(state.data_ALU.RAM[3], 8);
     printf("   D ==> RAM  3   (D3)         %s\n", hex_string);
     hex(state.data_ALU.RAM[4], 8);
     printf("   E ==> RAM  4   (D4)         %s\n", hex_string);
     hex(state.data_ALU.RAM[5], 8);
     printf("   F ==> RAM  5   (D5)         %s\n", hex_string);
     hex(state.data_ALU.RAM[6], 8);
     printf("   G ==> RAM  6   (D6)         %s\n", hex_string);
     hex(state.data_ALU.RAM[7], 8);
     printf("   H ==> RAM  7   (D7)         %s\n", hex_string);
     hex(state.data_ALU.RAM[8], 8);
     printf("   I ==> RAM  8   (OPERAND)    %s\n", hex_string);
     hex(state.data_ALU.RAM[9], 8);
     printf("   J ==> RAM  9   (WORK1)      %s\n", hex_string);
     hex(state.data_ALU.RAM[10], 8);
     printf("   K ==> RAM 10   (WORK2)      %s\n", hex_string);
     hex(state.data_ALU.RAM[11], 8);
     printf("   L ==> RAM 11   (WORK3)      %s\n", hex_string);
     hex(state.data_ALU.RAM[12], 8);
     printf("   M ==> RAM 12   (WORK4)      %s\n", hex_string);
     hex(state.data_ALU.RAM[13], 8);
     printf("   N ==> RAM 13   (WORK5)      %s\n", hex_string);
     hex(state.data_ALU.RAM[14], 8);
     printf("   O ==> RAM 14   (WORK6)      %s\n", hex_string);
     hex(state.data_ALU.RAM[15], 8);
     printf("   P ==> RAM 15   (WORK7)      %s\n", hex_string);
     hex(state.data_ALU.Q, 8);
     printf("   Q ==> Q Reg                 %s\n", hex_string);
     printf("   R ==> Exit\n\n");
     printf("Enter option: ");
     return(get_option('r'));
}
```

```
/*************************************************************************/
/* addr_ALU_state_menu:                                                  */
/*************************************************************************/

addr_ALU_state_menu()
{
     printf("\n\n");
     printf("=================================================\n\n");
     printf("Address ALU Menu\n\n");
     hex(state.addr_ALU.RAM[0], 8);
     printf("   A ==> RAM  0   (A0)          %s\n", hex_string);
     hex(state.addr_ALU.RAM[1], 8);
     printf("   B ==> RAM  1   (A1)          %s\n", hex_string);
     hex(state.addr_ALU.RAM[2], 8);
     printf("   C ==> RAM  2   (A2)          %s\n", hex_string);
     hex(state.addr_ALU.RAM[3], 8);
     printf("   D ==> RAM  3   (A3)          %s\n", hex_string);
     hex(state.addr_ALU.RAM[4], 8);
     printf("   E ==> RAM  4   (A4)          %s\n", hex_string);
     hex(state.addr_ALU.RAM[5], 8);
     printf("   F ==> RAM  5   (A5)          %s\n", hex_string);
     hex(state.addr_ALU.RAM[6], 8);
     printf("   G ==> RAM  6   (A6)          %s\n", hex_string);
     hex(state.addr_ALU.RAM[7], 8);
     printf("   H ==> RAM  7   (A7)          %s\n", hex_string);
     hex(state.addr_ALU.RAM[8], 8);
     printf("   I ==> RAM  8   (PC)          %s\n", hex_string);
     hex(state.addr_ALU.RAM[9], 8);
     printf("   J ==> RAM  9   (A7')         %s\n", hex_string);
     hex(state.addr_ALU.RAM[10], 8);
     printf("   K ==> RAM 10   (A7'')        %s\n", hex_string);
     hex(state.addr_ALU.RAM[11], 8);
     printf("   L ==> RAM 11   (DISPL)       %s\n", hex_string);
     hex(state.addr_ALU.RAM[12], 8);
     printf("   M ==> RAM 12   (INDEX)       %s\n", hex_string);
     hex(state.addr_ALU.RAM[13], 8);
     printf("   N ==> RAM 13   (MEM_PTR)     %s\n", hex_string);
     hex(state.addr_ALU.RAM[14], 8);
     printf("   O ==> RAM 14   (WORK8)       %s\n", hex_string);
     hex(state.addr_ALU.RAM[15], 8);
     printf("   P ==> RAM 15   (WORK9)       %s\n", hex_string);
     hex(state.addr_ALU.Q, 8);
     printf("   Q ==> Q Reg                  %s\n", hex_string);
     printf("   R ==> Exit\n\n");
     printf("Enter option: ");
     return(get_option('r'));
}
```

```
/*************************************************************************/
/* TCC_state_menu:                                                       */
/*************************************************************************/

TCC_state_menu()
{
     printf("\n\n");
     printf("=========================================================\n\n");
     printf("Current target flags:\n");
     printf("   C = %d   Z = %d   V = %d   N = %d   X = %d\n\n",
            state.TCC.C,state.TCC.Z,state.TCC.V,
            state.TCC.N,state.TCC.X);
     printf("Target Condition Codes Menu\n\n");
     printf("   A ==> Set C\n");
     printf("   B ==> Clear C\n");
     printf("   C ==> Set Z\n");
     printf("   D ==> Clear Z\n");
     printf("   E ==> Set V\n");
     printf("   F ==> Clear V\n");
     printf("   G ==> Set N\n");
     printf("   H ==> Clear N\n");
     printf("   I ==> Set X\n");
     printf("   J ==> Clear X\n");
     printf("   K ==> Exit\n\n");
     printf("Enter option: ");
     return(get_option('k'));
}


/*************************************************************************/
/* op_size_state_menu:                                                   */
/*************************************************************************/

op_size_state_menu()
{
     printf("\n\n");
     printf("=========================================================\n\n");
     switch(state.op_size) {
          case 0:
               printf("Current operand size = byte\n\n");
               break;
          case 1:
               printf("Current operand size = word\n\n");
               break;
          case 2:
               printf("Current operand size = long word\n\n");
               break;
     };
     printf("Operand Size Menu\n\n");
     printf("   A ==> Byte\n");
     printf("   B ==> Word\n");
     printf("   C ==> Long Word\n\n");
     printf("Enter option: ");
     return(get_option('c'));
}
```

```
/*************************************************************************/
/* mult_state_menu:                                                      */
/*************************************************************************/

mult_state_menu()
{
     printf("\n\n");
     printf("=========================================================\n\n");
     printf("Multiplier Menu\n\n");
     hex(state.mult.data_1, 8);
     printf("   A ==> Data 1            %s\n", hex_string);
     hex(state.mult.data_2, 8);
     printf("   B ==> Data 2            %s\n", hex_string);
     printf("   C ==> Exit\n\n");
     printf("Enter option: ");
     return(get_option('c'));
}


/*************************************************************************/
/* shifter_state_menu:                                                   */
/*************************************************************************/

shifter_state_menu()
{
     printf("\n\n");
     printf("=========================================================\n\n");
     printf("Shifter Menu\n\n");
     hex(state.shifter.data, 8);
     printf("   A ==> Data        %s\n", hex_string);
     hex(state.shifter.shift, 2);
     printf("   B ==> Shift             %s\n", hex_string);
     printf("   C ==> C                  %d\n", state.shifter.C);
     printf("   D ==> X                  %d\n", state.shifter.X);
     printf("   E ==> Z                  %d\n", state.shifter.Z);
     printf("   F ==> N                  %d\n", state.shifter.N);
     printf("   G ==> Exit\n\n");
     printf("Enter option: ");
     return(get_option('g'));
}
```

```
/*****************************************************************************/
/* useq_state_menu:                                                          */
/*****************************************************************************/

useq_state_menu()
{
     int i = 0;

     printf("\n\n");
     printf("=============================================================\n\n");
     printf("Current uPC stack:\n\n");
     while (i != state.useq.stack.top) {
          hex(state.useq.stack.element[i], 3);
          if (i == state.useq.stack.top - 1)
               printf("   TOP -->  %s\n\n", hex_string);
          else
               printf("            %s\n", hex_string);
          i++;
     };
     printf("Microsequencer Menu\n\n");
     hex(state.useq.uPC, 3);
     printf("   A ==> uPC                    %s\n", hex_string);
     printf("   B ==> Push to Stack\n");
     printf("   C ==> Pop from Stack\n");
     printf("   D ==> Exit\n\n");
     printf("Enter option: ");
     return(get_option('d'));
}
```

```
/*****************************************************************************/
/* uCS_menu1:                                                                */
/*****************************************************************************/

uCS_menu1()
{
     int uPC;

     uPC = state.useq.uPC;

     printf("\n\n");
     printf("=========================================================\n\n");
     hex (uPC, 3);
     printf("Microcontrol Store Menu  (uPC = %s)\n\n", hex_string);
     hex(uCS[uPC].useq.NA, 3);
     printf("   A ==> useq.NA              %s\n", hex_string);
     printf("   B ==> useq.inst              %x\n", uCS[uPC].useq.inst);
     printf("   C ==> useq.map_sel           %x\n", uCS[uPC].useq.map_sel);
     printf("   D ==> useq.CC_logic_level    %x\n",
            uCS[uPC].useq.CC_logic_level);
     hex(uCS[uPC].useq.CC_sel, 2);
     printf("   E ==> useq.CC_sel           %s\n", hex_string);
     printf("   F ==> IR_load                %x\n", uCS[uPC].IR_load);
     printf("   G ==> EWR_load               %x\n", uCS[uPC].EWR_load);
     printf("   H ==> op_size.load           %x\n", uCS[uPC].op_size.load);
     printf("   I ==> op_size.select         %x\n",
            uCS[uPC].op_size.select);
     printf("   J ==> op_size.data           %x\n", uCS[uPC].op_size.data);
     printf("   K ==> UIDB_sel               %x\n", uCS[uPC].UIDB_sel);
     printf("   L ==> More options\n");
     printf("   M ==> Exit\n\n");
     printf("Enter option: ");
     return(get_option('m'));
}
```

```
/*************************************************************************/
/* uPC_inst_PL_menu:                                                     */
/*************************************************************************/

uPC_inst_PL_menu()
{
     printf("\n\n");
     printf("=====================================================\n\n");
     printf("Current uPC_inst = ");
     switch (uCS[state.useq.uPC].useq.inst) {
          case 0:
               printf("JUMP ZERO\n\n");
               break;
          case 1:
               printf("COND JSB PL\n\n");
               break;
          case 2:
               printf("JUMP MAP\n\n");
               break;
          case 3:
               printf("COND JUMP PL\n\n");
               break;
          case 4:
               printf("PUSH/COND LD CNTR\n\n");
               break;
          case 5:
               printf("COND JSB R/PL\n\n");
               break;
          case 6:
               printf("COND JUMP VECT\n\n");
               break;
          case 7:
               printf("COND JUMP R/PL\n\n");
               break;
          case 8:
               printf("REPEAT LOOP, CNTR <> 0\n\n");
               break;
          case 9:
               printf("REPEAT PL, CNTR <> 0\n\n");
               break;
          case 10:
               printf("COND RTN\n\n");
               break;
          case 11:
               printf("COND JUMP PL AND POP\n\n");
               break;
          case 12:
               printf("LD CNTR AND CONT\n\n");
               break;
          case 13:
               printf("TEST END LOOP\n\n");
               break;
          case 14:
               printf("CONT\n\n");
               break;
```

```
        case 15:
             printf("3 - WAY BRANCH\n\n");
             break;
    };
    printf("Microprogram counter instruction Menu\n\n");
    printf("   A ==> JUMP ZERO\n");
    printf("   B ==> COND JSB PL\n");
    printf("   C ==> JUMP MAP\n");
    printf("   D ==> COND JUMP PL\n");
    printf("   E ==> PUSH/COND LD CNTR\n");
    printf("   F ==> COND JSB R/PL\n");
    printf("   G ==> COND JUMP VECT\n");
    printf("   H ==> COND JUMP R/PL\n");
    printf("   I ==> REPEAT LOOP, CNTR <> 0\n");
    printf("   J ==> REPEAT PL, CNTR <> 0\n");
    printf("   K ==> COND RTN\n");
    printf("   L ==> COND JUMP PL AND POP\n");
    printf("   M ==> LD CNTR AND CONT\n");
    printf("   N ==> TEST END LOOP\n");
    printf("   O ==> CONT\n");
    printf("   P ==> 3 - WAY BRANCH\n\n");
    printf("Enter option: ");
    return(get_option('p'));
}
```

```
/*************************************************************************/
/* uPC_map_sel_PL_menu:                                                  */
/*************************************************************************/

uPC_map_sel_PL_menu()
{
     printf("\n\n");
     printf("=============================================\n\n");
     printf("Current useq.map_sel = ");
     switch (uCS[state.useq.uPC].useq.map_sel) {
          case 0:
               printf("PL Next Address\n\n");
               break;
          case 1:
               printf("EA PROM\n\n");
               break;
          case 2:
               printf("IR PROM\n\n");
               break;
     };
     printf("Microsequencer Mapping PROM Menu\n\n");
     printf("    A ==> PL Next Address\n");
     printf("    B ==> EA PROM\n");
     printf("    C ==> IR PROM\n\n");
     printf("Enter option: ");
     return(get_option('c'));
}


/*************************************************************************/
/* op_size_sel_PL_menu:                                                  */
/*************************************************************************/

op_size_sel_PL_menu()
{
     printf("\n\n");
     printf("=====================================================\n\n");
     switch (uCS[state.useq.uPC].op_size.select) {
          case 0:
               printf("Current operand size select = PL\n\n");
               break;
          case 1:
               printf("Current operand size select = IR(7-6)\n\n");
               break;
          case 2:
               printf("Current operand size select = IR(10-9)\n\n");
               break;
          case 3:
               printf("Current operand size select = IR(13-12)\n\n");
               break;
     };
     printf("Operand Size Select Menu\n\n");
     printf("    A ==> PL\n");
     printf("    B ==> IR(7-6)\n");
     printf("    C ==> IR(10-9)\n");
     printf("    D ==> IR(13-12)\n\n");
     printf("Enter option: ");
     return(get_option('d'));
}
```

```
/*************************************************************************/
/* op_size_PL_menu:                                                      */
/*************************************************************************/

op_size_PL_menu()
{
     printf("\n\n");
     printf("=========================================================\n\n");
     switch(uCS[state.useq.uPC].op_size.data) {
          case 0:
               printf("Current operand size = byte\n\n");
               break;
          case 1:
               printf("Current operand size = word\n\n");
               break;
          case 2:
               printf("Current operand size = long word\n\n");
               break;
     };
     printf("Operand Size Menu\n\n");
     printf("   A ==> Byte\n");
     printf("   B ==> Word\n");
     printf("   C ==> Long Word\n\n");
     printf("Enter option: ");
     return(get_option('c'));
}
```

```
/*************************************************************************/
/* UIDB_sel_menu:                                                        */
/*************************************************************************/

UIDB_sel_menu()
{
     printf("\n\n");
     printf("=====================================================\n\n");
     switch(uCS[state.useq.uPC].UIDB_sel) {
          case 0:
               printf("Current UIDB device = No device selected\n\n");
               break;
          case 1:
               printf("Current UIDB device = DBI\n\n");
               break;
          case 2:
               printf("Current UIDB device = Constants PROM\n\n");
               break;
          case 3:
               printf("Current UIDB device = op_size offset PROM\n\n");
               break;
          case 4:
               printf("Current UIDB device = Target SR\n\n");
               break;
          case 5:
               printf("Current UIDB device = EWR bits 7-0\n\n");
               break;
          case 6:
               printf("Current UIDB device = Feedback latch\n\n");
               break;
          case 7:
               printf("Current UIDB device = Shifter\n\n");
               break;
          case 8:
               printf("Current UIDB device = Lower mult. result\n\n");
               break;
          case 9:
               printf("Current UIDB device = Upper mult. result\n\n");
               break;
          case 10:
               printf("Current UIDB device = EWR bits 15-0\n\n");
               break;
          case 11:
               printf("Current UIDB device = IR bits 11-9\n\n");
               break;
          case 12:
               printf("Current UIDB device = IR bits 7-0\n\n");
               break;
     };
```

```
    printf("UIDB Device Select Menu\n\n");
    printf("    A ==> No device\n");
    printf("    B ==> DBI\n");
    printf("    C ==> Constants PROM\n");
    printf("    D ==> op_size offset PROM\n");
    printf("    E ==> Target SR\n");
    printf("    F ==> EWR bits 7-0\n");
    printf("    G ==> Feedback latch\n");
    printf("    H ==> Shifter\n");
    printf("    I ==> Lower multiplier result\n");
    printf("    J ==> Upper multiplier result\n");
    printf("    K ==> EWR bits 15-0\n");
    printf("    L ==> IR bits 11-9\n\n");
    printf("    M ==> IR bits 7-0\n");
    printf("Enter option: ");
    return(get_option('m'));
}
```

```
/*************************************************************************/
/* uCS_menu2:                                                            */
/*************************************************************************/

uCS_menu2()
{
     int uPC;

     uPC = state.useq.uPC;
     printf("\n\n");
     printf("=========================================================\n\n");
     hex (uPC, 3);
     printf("Microcontrol Store Menu  (uPC = %s)\n\n", hex_string);
     printf("   A ==> data_ALU inst      ");
     printf("%x ", uCS[uPC].data_ALU.dest);
     printf("%x ", uCS[uPC].data_ALU.funct);
     printf("%x\n", uCS[uPC].data_ALU.src);
     printf("   B ==> data_ALU.OE          %x\n",
            uCS[uPC].data_ALU.OE);
     printf("   C ==> data_ALU.A_bus       %x\n",
            uCS[uPC].data_ALU.A_bus);
     printf("   D ==> data_ALU.A_bus_sel   %x\n",
            uCS[uPC].data_ALU.A_bus_sel);
     printf("   E ==> data_ALU.B_bus       %x\n",
            uCS[uPC].data_ALU.B_bus);
     printf("   F ==> data_ALU.B_bus_sel   %x\n",
            uCS[uPC].data_ALU.B_bus_sel);
     printf("   G ==> addr_ALU inst      ");
     printf("%x ", uCS[uPC].addr_ALU.dest);
     printf("%x ", uCS[uPC].addr_ALU.funct);
     printf("%x\n", uCS[uPC].addr_ALU.src);
     printf("   H ==> addr_ALU.OE          %x\n",
            uCS[uPC].addr_ALU.OE);
     printf("   I ==> addr_ALU.A_bus       %x\n",
            uCS[uPC].addr_ALU.A_bus);
     printf("   J ==> addr_ALU.A_bus_sel   %x\n",
            uCS[uPC].addr_ALU.A_bus_sel);
     printf("   K ==> addr_ALU.B_bus       %x\n",
            uCS[uPC].addr_ALU.B_bus);
     printf("   L ==> addr_ALU.B_bus_sel   %x\n",
            uCS[uPC].addr_ALU.B_bus_sel);
     printf("   M ==> MAR.load             %x\n", uCS[uPC].MAR.load);
     hex(uCS[uPC].const_addr, 2);
     printf("   N ==> const_addr          %s\n", hex_string);
     printf("   O ==> More options\n");
     printf("   P ==> Exit\n\n");
     printf("Enter option: ");
     return(get_option('p'));
}
```

```
/*****************************************************************************/
/* addr_data_ALU_inst_PL_menu:                                               */
/*****************************************************************************/

addr_data_ALU_inst_PL_menu(s)
char *s;                        /* option string */
{
     printf("\n\n");
     printf("=========================================================\n\n");
     printf("%s ALU Instruction Menu\n\n", s);
     printf("   A ==> Change sources\n");
     printf("   B ==> Change function\n");
     printf("   C ==> Change destination\n");
     printf("   D ==> Enter hex instruction\n");
     printf("   E ==> Exit\n\n");
     printf("Enter option: ");
     return(get_option('e'));
}
```

```
/*************************************************************************/
/* addr_data_ALU_src_PL_menu:                                            */
/*************************************************************************/

addr_data_ALU_src_PL_menu(s)
char *s;                        /* option string */
{
     int source;                /* instruction source */

     printf("\n\n");
     printf("=====================================================\n\n");
     if (*s == 'A')
          source = uCS[state.useq.uPC].addr_ALU.src;
     else
          source = uCS[state.useq.uPC].data_ALU.src;
     switch (source) {
          case 0:
               printf("Current source = AQ\n\n");
               break;
          case 1:
               printf("Current source = AB\n\n");
               break;
          case 2:
               printf("Current source = ZQ\n\n");
               break;
          case 3:
               printf("Current source = ZB\n\n");
               break;
          case 4:
               printf("Current source = ZA\n\n");
               break;
          case 5:
               printf("Current source = DA\n\n");
               break;
          case 6:
               printf("Current source = DQ\n\n");
               break;
          case 7:
               printf("Current source = DZ\n\n");
               break;
     };
     printf("%s ALU Source Menu\n\n", s);
     printf("    A ==> AQ\n");
     printf("    B ==> AB\n");
     printf("    C ==> ZQ\n");
     printf("    D ==> ZB\n");
     printf("    E ==> ZA\n");
     printf("    F ==> DA\n");
     printf("    G ==> DQ\n");
     printf("    H ==> DZ\n\n");
     printf("Enter option: ");
     return(get_option('h'));
}
```

```
/*************************************************************************/
/* addr_data_ALU_funct_PL_menu:                                          */
/*************************************************************************/

addr_data_ALU_funct_PL_menu(s)
char *s;                        /* option string */
{
     int funct;                 /* instruction function */
     printf("\n\n");
     printf("=====================================================\n\n");
     if (*s == 'A')
          funct = uCS[state.useq.uPC].addr_ALU.funct;
     else
          funct = uCS[state.useq.uPC].data_ALU.funct;
     switch (funct) {
          case 0:
               printf("Current function = ADD\n\n");
               break;
          case 1:
               printf("Current function = SUBR\n\n");
               break;
          case 2:
               printf("Current function = SUBS\n\n");
               break;
          case 3:
               printf("Current function = OR\n\n");
               break;
          case 4:
               printf("Current function = AND\n\n");
               break;
          case 5:
               printf("Current function = NOTRS\n\n");
               break;
          case 6:
               printf("Current function = EXOR\n\n");
               break;
          case 7:
               printf("Current function = EXNOR\n\n");
               break;
     };
     printf("%s ALU Function Menu\n\n", s);
     printf("    A ==> ADD\n");
     printf("    B ==> SUBR\n");
     printf("    C ==> SUBS\n");
     printf("    D ==> OR\n");
     printf("    E ==> AND\n");
     printf("    F ==> NOTRS\n");
     printf("    G ==> EXOR\n");
     printf("    H ==> EXNOR\n\n");
     printf("Enter option: ");
     return(get_option('h'));
}
```

```
/*************************************************************************/
/* addr_data_ALU_dest_PL_menu:                                           */
/*************************************************************************/

addr_data_ALU_dest_PL_menu(s)
char *s;                        /* option string */
{
     int dest;                  /* instruction destination */

     printf("\n\n");
     printf("=====================================================\n\n");
     if (*s == 'A')
          dest = uCS[state.useq.uPC].addr_ALU.dest;
     else
          dest = uCS[state.useq.uPC].data_ALU.dest;
     switch (dest) {
          case 0:
               printf("Current destination = QREG\n\n");
               break;
          case 1:
               printf("Current destination = NOP\n\n");
               break;
          case 2:
               printf("Current destination = RAMA\n\n");
               break;
          case 3:
               printf("Current destination = RAMF\n\n");
               break;
          case 4:
               printf("Current destination = RAMQD\n\n");
               break;
          case 5:
               printf("Current destination = RAMD\n\n");
               break;
          case 6:
               printf("Current destination = RAMQU\n\n");
               break;
          case 7:
               printf("Current destination = RAMU\n\n");
               break;
     };
     printf("%s ALU Destination Menu\n\n", s);
     printf("    A ==> QREG\n");
     printf("    B ==> NOP\n");
     printf("    C ==> RAMA\n");
     printf("    D ==> RAMF\n");
     printf("    E ==> RAMQD\n");
     printf("    F ==> RAMD\n");
     printf("    G ==> RAMQU\n");
     printf("    H ==> RAMU\n\n");
     printf("Enter option: ");
     return(get_option('h'));
}
```

```
/*************************************************************************/
/* A_B_bus_sel_PL_menu:                                                  */
/*************************************************************************/

A_B_bus_sel_PL_menu(ALU,bus)
char *ALU;                      /* ALU option string */
char bus;                       /* bus option character */
{
     int bus_sel;               /* temp bus selection */

     printf("\n\n");
     printf("=========================================================\n\n");
     if ((*ALU == 'A') && (bus == 'A'))
          bus_sel = uCS[state.useq.uPC].addr_ALU.A_bus_sel;
     else if ((*ALU == 'A') && (bus == 'B'))
          bus_sel = uCS[state.useq.uPC].addr_ALU.B_bus_sel;
     else if ((*ALU == 'D') && (bus == 'A'))
          bus_sel = uCS[state.useq.uPC].data_ALU.A_bus_sel;
     else if ((*ALU == 'D') && (bus == 'B'))
          bus_sel = uCS[state.useq.uPC].data_ALU.B_bus_sel;

     switch (bus_sel) {
          case 0:
               printf("Current %s_ALU.%c_bus_sel = PL\n\n",
                      ALU, bus);
               break;
          case 1:
               printf("Current %s_ALU.%c_bus_sel = IR (2-0)\n\n",
                      ALU, bus);
               break;
          case 2:
               printf("Current %s_ALU.%c_bus_sel = IR (8-6)\n\n",
                      ALU, bus);
               break;
          case 3:
               printf("Current %s_ALU.%c_bus_sel = IR (11-9)\n\n",
                      ALU, bus);
               break;
          case 4:
               printf("Current %s_ALU.%c_bus_sel = IR (14-12)\n\n",
                      ALU, bus);
               break;
          case 5:
               printf("Current %s_ALU.%c_bus_sel = EWR (14-12)\n\n",
                      ALU, bus);
               break;
          case 6:
               printf("Current %s_ALU.%c_bus_sel = EAR (2-0)\n\n",
                      ALU, bus);
               break;
          case 7:
               printf("Current %s_ALU.%c_bus_sel = EWR (2-0)\n\n",
                      ALU, bus);
               break;
     };
```

```
        printf("%s ALU %c Bus Select Menu\n\n", ALU, bus);
        printf("    A ==> PL\n");
        printf("    B ==> IR (2-0)\n");
        printf("    C ==> IR (8-6)\n");
        printf("    D ==> IR (11-9)\n");
        printf("    E ==> IR (14-12)\n");
        printf("    F ==> EWR (14-12)\n");
        printf("    G ==> EAR (2-0)\n\n");
        printf("    H ==> EWR (2-0)\n\n");
        printf("Enter option: ");
        return(get_option('h'));
}


/*************************************************************************/
/* uCS_menu3:                                                            */
/*************************************************************************/

uCS_menu3()
{
        int uPC;

        uPC = state.useq.uPC;

        printf("\n\n");
        printf("=====================================================\n\n");
        hex (uPC, 3);
        printf("Microcontrol Store Menu  (uPC = %s)\n\n", hex_string);
        printf("    A ==> DBI.load          %x\n", uCS[uPC].DBI.load);
        printf("    B ==> MM.select         %x\n", uCS[uPC].MM.select);
        printf("    C ==> MM.read           %x\n", uCS[uPC].MM.read);
        printf("    D ==> MM.size_sel       %x\n", uCS[uPC].MM.size_sel);
        printf("    E ==> MM.size           %x\n", uCS[uPC].MM.size);
        printf("    F ==> TCC.X_load        %x\n", uCS[uPC].TCC.X_load);
        printf("    G ==> TCC.N_load        %x\n", uCS[uPC].TCC.N_load);
        printf("    H ==> TCC.Z_load        %x\n", uCS[uPC].TCC.Z_load);
        printf("    I ==> TCC.V_load        %x\n", uCS[uPC].TCC.V_load);
        printf("    J ==> TCC.C_load        %x\n", uCS[uPC].TCC.C_load);
        printf("    K ==> SR_sel            %x\n", uCS[uPC].SR_sel);
        printf("    L ==> Cin.data          %x\n", uCS[uPC].Cin.data);
        printf("    M ==> Cin.select        %x\n", uCS[uPC].Cin.select);
        printf("    N ==> More options\n");
        printf("    O ==> Exit\n\n");
        printf("Enter option: ");
        return(get_option('o'));
}
```

```
/*************************************************************************/
/* MM_size_sel_PL_menu:                                                  */
/*************************************************************************/

MM_size_sel_PL_menu()
{
     printf("\n\n");
     printf("=========================================================\n\n");
     switch (uCS[state.useq.uPC].MM.size_sel) {
          case 0:
               printf("Current MM_size_sel = PL\n\n");
               break;
          case 1:
               printf("Current MM_size_sel = op_size reg\n\n");
               break;
          case 2:
               printf("Current MM_size_sel = EWR(1-0)\n\n");
               break;
          case 3:
               printf("Current MM_size_sel = EWR(5-4)\n\n");
               break;
     }
     printf("MM Size Select Menu\n\n");
     printf("    A ==> PL\n");
     printf("    B ==> op_size reg\n");
     printf("    C ==> EWR(1-0)\n");
     printf("    D ==> EWR(5-4)\n\n");
     printf("Enter option: ");
     return(get_option('d'));
}


/*************************************************************************/
/* MM_size_PL_menu:                                                      */
/*************************************************************************/

MM_size_PL_menu()
{
     printf("\n\n");
     printf("=========================================================\n\n");
     switch (uCS[state.useq.uPC].MM.size) {
          case 0:
               printf("Current MM_size = Byte\n\n");
               break;
          case 1:
               printf("Current MM_size = Word\n\n");
               break;
          case 2:
               printf("Current MM_size = Long Word\n\n");
               break;
     }
     printf("MM Size Menu\n\n");
     printf("    A ==> Byte\n");
     printf("    B ==> Word\n");
     printf("    C ==> Long Word\n\n");
     printf("Enter option: ");
     return(get_option('c'));
}
```

```
/*************************************************************************/
/* SR_sel_PL_menu:                                                       */
/*************************************************************************/

SR_sel_PL_menu()
{
     printf("\n\n");
     printf("=====================================================\n\n");
     switch (uCS[state.useq.uPC].SR_sel) {
          case 0:
               printf("Current SR_sel = Data ALU\n\n");
               break;
          case 1:
               printf("Current SR_sel = UIDB\n\n");
               break;
          case 2:
               printf("Current SR_sel = Shifter\n\n");
               break;
          case 3:
               printf("Current SR_sel = Address ALU\n\n");
               break;
     }
     printf("SR Select Menu\n\n");
     printf("   A ==> Data ALU\n");
     printf("   B ==> UIDB\n");
     printf("   C ==> Shifter\n");
     printf("   D ==> Address ALU\n\n");
     printf("Enter option: ");
     return(get_option('d'));
}


/*************************************************************************/
/* Cin_sel_PL_menu:                                                      */
/*************************************************************************/

Cin_sel_PL_menu()
{
     printf("\n\n");
     printf("=====================================================\n\n");
     switch (uCS[state.useq.uPC].Cin.select) {
          case 0:
               printf("Current Cin.select = PL\n\n");
               break;
          case 1:
               printf("Current Cin.select = Status register\n\n");
               break;
     }
     printf("Cin Select Menu\n\n");
     printf("   A ==> PL\n");
     printf("   B ==> Status register\n\n");
     printf("Enter option: ");
     return(get_option('b'));
}
```

```
/*************************************************************************/
/* uCS_menu4:                                                            */
/*************************************************************************/

uCS_menu4()
{
     int uPC;

     uPC = state.useq.uPC;

     printf("\n\n");
     printf("=============================================================\n\n");
     hex (uPC, 3);
     printf("Microcontrol Store Menu  (uPC = %s)\n\n", hex_string);
     printf("   A ==> mult.data_1_load          %x\n",
            uCS[uPC].mult.data_1_load);
     printf("   B ==> mult.data_2_load          %x\n",
            uCS[uPC].mult.data_2_load);
     printf("   C ==> shifter.inst              %x\n",
            uCS[uPC].shifter.inst);
     printf("   D ==> shifter.shift_load        %x\n",
            uCS[uPC].shifter.shift_load);
     printf("   E ==> shifter.data_load         %x\n",
            uCS[uPC].shifter.data_load);
     printf("   F ==> shifter.CX_load           %x\n",
            uCS[uPC].shifter.CX_load);
     printf("   G ==> feedback.load             %x\n",
            uCS[uPC].feedback.load);
     printf("   H ==> feedback.size_sel         %x\n",
            uCS[uPC].feedback.size_sel);
     printf("   I ==> feedback.size             %x\n",
            uCS[uPC].feedback.size);
     printf("   J ==> feedback.scale_sel        %x\n",
            uCS[uPC].feedback.scale_sel);
     printf("   K ==> EAR.load                  %x\n",
            uCS[uPC].EAR.load);
     printf("   L ==> EAR.select                %x\n",
            uCS[uPC].EAR.select);
     printf("   M ==> data_ALU_mask             %x\n",
            uCS[uPC].data_ALU_mask);
     printf("   N ==> More options\n");
     printf("   O ==> Exit\n\n");
     printf("Enter option: ");
     return(get_option('o'));
}
```

```
/*************************************************************************/
/* shifter_inst_PL_menu:                                                 */
/*************************************************************************/

shifter_inst_PL_menu()
{
     printf("\n\n");
     printf("=========================================================\n\n");
     switch (uCS[state.useq.uPC].shifter.inst) {
          case 0:
               printf("Current shifter.inst = ASR\n\n");
               break;
          case 1:
               printf("Current shifter.inst = ASL\n\n");
               break;
          case 2:
               printf("Current shifter.inst = LSR\n\n");
               break;
          case 3:
               printf("Current shifter.inst = LSL\n\n");
               break;
          case 4:
               printf("Current shifter.inst = ROXR\n\n");
               break;
          case 5:
               printf("Current shifter.inst = ROXL\n\n");
               break;
          case 6:
               printf("Current shifter.inst = ROR\n\n");
               break;
          case 7:
               printf("Current shifter.inst = ROL\n\n");
               break;
     }
     printf("Shifter Instruction Menu\n\n");
     printf("    A ==> ASR\n");
     printf("    B ==> ASL\n");
     printf("    C ==> LSR\n");
     printf("    D ==> LSL\n");
     printf("    E ==> ROXR\n");
     printf("    F ==> ROXL\n");
     printf("    G ==> ROR\n");
     printf("    H ==> ROL\n\n");
     printf("Enter option: ");
     return(get_option('h'));
}
```

```
/*************************************************************************/
/* feedback_size_sel_PL_menu:                                            */
/*************************************************************************/

feedback_size_sel_PL_menu()
{
     printf("\n\n");
     printf("=====================================================\n\n");
     switch (uCS[state.useq.uPC].feedback.size_sel) {
          case 0:
               printf("Current size_sel = PL\n\n");
               break;
          case 1:
               printf("Current size_sel = op_size\n\n");
               break;
          case 2:
               printf("Current size_sel = EWR bit 11\n\n");
               break;
     }
     printf("Feedback Sign Extension Size Selector Menu\n\n");
     printf("   A ==> PL\n");
     printf("   B ==> op_size\n");
     printf("   C ==> EWR bit 11\n\n");
     printf("Enter option: ");
     return(get_option('c'));
}


/*************************************************************************/
/* feedback_size_PL_menu:                                                */
/*************************************************************************/

feedback_size_PL_menu()
{
     printf("\n\n");
     printf("=====================================================\n\n");
     switch (uCS[state.useq.uPC].feedback.size) {
          case 0:
               printf("Current size = Byte\n\n");
               break;
          case 1:
               printf("Current size = Word\n\n");
               break;
          case 2:
               printf("Current size = Long word\n\n");
               break;
     }
     printf("Feedback Size Selector Menu\n\n");
     printf("   A ==> Byte\n");
     printf("   B ==> Word\n");
     printf("   C ==> Long word\n\n");
     printf("Enter option: ");
     return(get_option('c'));
}
```

```
/*************************************************************************/
/* breaks_menu:                                                          */
/*************************************************************************/

breaks_menu()
{
     printf("\n\n");
     printf("=============================================\n\n");
     printf("Breakpoint menu\n\n");
     printf("   A ==> Set the breakpoint\n");
     printf("   B ==> Clear the breakpoint\n\n");
     printf("Enter option: ");
     return(get_option('b'));
}


/*************************************************************************/
/* get_option:                                                           */
/*   Function that gets a menu choice from STDIN and returns it.  If     */
/*   the user selects a menu option that has an ascii value less than    */
/*   'a' or more than limit, then the routine will wait for another      */
/*   selection from the user.                                            */
/*************************************************************************/

get_option(limit)
char limit;                    /* character limit for user response */
{
     char ch;                  /* user option selection */

     do {
          ch = getchar();
          ch = tolower(ch);
     } while ((ch < 'a') || (limit < ch));
     while (getchar() != '\n')
          ;
     return(ch);
}


/*************************************************************************/
/* hex:                                                                  */
/*      Routine that takes the integer i and converts it to a hex        */
/*      string with j hex digits.  The string is returned in the         */
/*      global variable hex_string.                                      */
/*************************************************************************/

hex(i, j)
int i;                         /* integer to convert to hex */
int j;                         /* maximum number of bytes */
{
     char s[10];               /* string array */

     hex_string[0] = ' ';
     sprintf(s, "%x", i);
     for (i=strlen(s);i<j;i++)
          strcat(hex_string,"0");
     strcat(hex_string,s);
}
```

```
/*****************************************************************************/
/* change.c                                                                  */
/*                                                                           */
/*      Change.c is a set of functions that are responsible for              */
/*      changing the state of the host.  All routines follow the same        */
/*      format - they call a routine that prints a menu and gets the         */
/*      user's option selection, and then go into a switch statement         */
/*      that executes the option.  Due to this 'cut and dried'               */
/*      algorithm used by almost all of the below procedures, putting        */
/*      descriptive function headers on each function would be tedious       */
/*      to the programmer and useless to the person browsing over the        */
/*      program on a leisurely Sunday afternoon.  Thus, only                 */
/*      procedures that warrant special attention will have function         */
/*      headers.                                                             */
/*                                                                           */
/*      NOTE:  It may be helpful for anyone looking over this portion        */
/*             of code to have a copy of the menu routines close by.         */
/*             This will let you see what each option in the below           */
/*             switch statement does.                                        */
/*****************************************************************************/
```

```
/*****************************************************************/
/* change_state:                                                 */
/*****************************************************************/

change_state()
{
     char option;              /* user option */

     do {
          option = change_state_menu();
          switch (option) {
               case 'a':
                    change_state_data_ALU();
                    break;
               case 'b':
                    change_state_addr_ALU();
                    break;
               case 'c':
                    printf("\n\n");
                    printf("New DBI.data = ");
                    scanf("%x", &state.DBI);
                    break;
               case 'd':
                    printf("\n\n");
                    printf("New MAR.data = ");
                    scanf("%x", &state.MAR);
                    break;
               case 'e':
                    change_state_TCC();
                    break;
               case 'f':
                    change_state_op_size();
                    break;
               case 'g':
                    printf("\n\n");
                    printf("New PIR = ");
                    scanf("%x", &state.PIR);
                    break;
               case 'h':
                    printf("\n\n");
                    printf("New IR = ");
                    scanf("%x", &state.IR);
                    break;
               case 'i':
                    printf("\n\n");
                    printf("New EWR = ");
                    scanf("%x", &state.EWR);
                    break;
               case 'j':
                    change_state_mult();
                    break;
               case 'k':
                    change_state_shifter();
                    break;
               case 'l':
                    printf("\n\n");
                    printf("New feedback = ");
                    scanf("%x", &state.feedback);
                    break;
```

```
                case 'm':
                     printf("\n\n");
                     printf("New EAR = ");
                     scanf("%x", &state.EAR);
                     break;
                case 'n':
                     printf("\n\n");
                     printf("New LIDB register = ");
                     scanf("%x", &state.LIDB);
                     break;
                case 'o':
                     change_state_useq();
                     break;
                case 'p':
                     change_uCS1();
                     break;
          };
     } while (option != 'q');
}
```

```
/*****************************************************************************/
/* change_state_data_ALU:                                                    */
/*****************************************************************************/

change_state_data_ALU()
{
     int i;                    /* RAM array index */
     char option;              /* user option */

     do {
          option = data_ALU_state_menu();
          switch (option) {
               case 'a':
               case 'b':
               case 'c':
               case 'd':
               case 'e':
               case 'f':
               case 'g':
               case 'h':
               case 'i':
               case 'j':
               case 'k':
               case 'l':
               case 'm':
               case 'n':
               case 'o':
               case 'p':
                    i = option - 'a';
                    printf("\n\n");
                    printf("New RAM %d = ", i);
                    scanf("%x", &state.data_ALU.RAM[i]);
                    break;
               case 'q':
                    printf("\n\n");
                    printf("New Q REG = ");
                    scanf("%x", &state.data_ALU.Q);
                    break;
          };
     } while (option != 'r');
}
```

```
/*****************************************************************************/
/* change_state_addr_ALU:                                                    */
/*****************************************************************************/

change_state_addr_ALU()
{
     int i;                    /* RAM array index */
     char option;              /* user option */

     do {
          option = addr_ALU_state_menu();
          switch (option) {
               case 'a':
               case 'b':
               case 'c':
               case 'd':
               case 'e':
               case 'f':
               case 'g':
               case 'h':
               case 'i':
               case 'j':
               case 'k':
               case 'l':
               case 'm':
               case 'n':
               case 'o':
               case 'p':
                    i = option - 'a';
                    printf("\n\n");
                    printf("New RAM %d = ", i);
                    scanf("%x", &state.addr_ALU.RAM[i]);
                    break;
               case 'q':
                    printf("\n\n");
                    printf("New Q REG = ");
                    scanf("%x", &state.addr_ALU.Q);
                    break;
          };
     } while (option != 'r');
}
```

```
/*************************************************************************/
/* change_state_TCC:                                                     */
/*************************************************************************/

change_state_TCC()
{
     char option;              /* user option */

     do {
          option = TCC_state_menu();
          switch (option) {
               case 'a':
                    state.TCC.C = 1;
                    break;
               case 'b':
                    state.TCC.C = 0;
                    break;
               case 'c':
                    state.TCC.Z = 1;
                    break;
               case 'd':
                    state.TCC.Z = 0;
                    break;
               case 'e':
                    state.TCC.V = 1;
                    break;
               case 'f':
                    state.TCC.V = 0;
                    break;
               case 'g':
                    state.TCC.N = 1;
                    break;
               case 'h':
                    state.TCC.N = 0;
                    break;
               case 'i':
                    state.TCC.X = 1;
                    break;
               case 'j':
                    state.TCC.X = 0;
                    break;
          };
     } while (option != 'k');
}
```

```
/*****************************************************************************/
/* change_state_op_size:                                                     */
/*****************************************************************************/

change_state_op_size()
{
     char option;              /* user option */

     option = op_size_state_menu();
     switch (option) {
          case 'a':
               state.op_size = 0;
               break;
          case 'b':
               state.op_size = 1;
               break;
          case 'c':
               state.op_size = 2;
               break;
     };
}


/*****************************************************************************/
/* change_state_mult:                                                        */
/*****************************************************************************/

change_state_mult()
{
     char option;              /* user option */

     do {
          option = mult_state_menu();
          switch (option) {
               case 'a':
                    printf("\n\n");
                    printf("New Data 1 = ");
                    scanf("%x", &state.mult.data_1);
                    break;
               case 'b':
                    printf("\n\n");
                    printf("New Data 2 = ");
                    scanf("%x", &state.mult.data_2);
                    break;
          };
     } while (option != 'c');
}
```

```
/*************************************************************************/
/* change_state_shifter:                                                 */
/*************************************************************************/

change_state_shifter()
{
     char option;              /* user option */

     do {
          option = shifter_state_menu();
          switch (option) {
               case 'a':
                    printf("\n\n");
                    printf("New Data = ");
                    scanf("%x", &state.shifter.data);
                    break;
               case 'b':
                    printf("\n\n");
                    printf("New Shift = ");
                    scanf("%x", &state.shifter.shift);
                    break;
               case 'c':
                    printf("\n\n");
                    printf("New C Register = ");
                    scanf("%x", &state.shifter.C);
                    break;
               case 'd':
                    printf("\n\n");
                    printf("New X Register = ");
                    scanf("%x", &state.shifter.X);
                    break;
               case 'e':
                    printf("\n\n");
                    printf("New Z Register = ");
                    scanf("%x", &state.shifter.Z);
                    break;
               case 'f':
                    printf("\n\n");
                    printf("New N Register = ");
                    scanf("%x", &state.shifter.N);
                    break;
          };
     } while (option != 'g');
}
```

```
/*****************************************************************************/
/* change_state_useq:                                                        */
/*****************************************************************************/

change_state_useq()
{
     int top;                    /* stack top pointer */
     char option;                /* user option */

     do {
          option = useq_state_menu();
          switch (option) {
               case 'a':
                    printf("\n\n");
                    printf("New uPC = ");
                    scanf("%x", &state.useq.uPC);
                    break;
               case 'b':
                    printf("\n\n");
                    printf("Stack data = ");
                    top = state.useq.stack.top;
                    if (top < 9) {
                         scanf("%x", &state.useq.stack.element[top]);
                         state.useq.stack.top++;
                    }
                    else
                         scanf("%x", &state.useq.stack.element[top-1]);
                    break;
               case 'c':
                    if (state.useq.stack.top > 0)
                         --state.useq.stack.top;
                    break;
          };
     } while (option != 'd');
}
```

```
/*************************************************************************/
/* change_uCS1:                                                          */
/*************************************************************************/

change_uCS1()
{
     int uPC;                    /* microprogram counter */
     char option;                /* user option */

     uPC = state.useq.uPC;
     do {
          option = uCS_menu1();
          switch (option) {
               case 'a':
                    printf("\n\n");
                    printf("New useq.NA = ");
                    scanf("%x", &uCS[uPC].useq.NA);
                    break;
               case 'b':
                    change_PL_uPC_inst();
                    break;
               case 'c':
                    change_PL_uPC_map_sel();
                    break;
               case 'd':
                    printf("\n\n");
                    printf("New useq.CC_logic_level = ");
                    scanf("%x", &uCS[uPC].useq.CC_logic_level);
                    break;
               case 'e':
                    printf("\n\n");
                    printf("New useq.CC_sel = ");
                    scanf("%x", &uCS[uPC].useq.CC_sel);
                    break;
               case 'f':
                    printf("\n\n");
                    printf("New IR_load = ");
                    scanf("%x", &uCS[uPC].IR_load);
                    break;
               case 'g':
                    printf("\n\n");
                    printf("New EWR_load = ");
                    scanf("%x", &uCS[uPC].EWR_load);
                    break;
               case 'h':
                    printf("\n\n");
                    printf("New op_size.load = ");
                    scanf("%x", &uCS[uPC].op_size.load);
                    break;
               case 'i':
                    change_PL_op_size_sel();
                    break;
               case 'j':
                    change_PL_op_size();
                    break;
               case 'k':
                    change_UIDB_sel();
                    break;
```

```
                case 'l':
                        change_uCS2();
                        option = 'm';
                        break;
                };
        } while (option != 'm');
}


/*************************************************************************/
/* change_PL_uPC_inst:                                                   */
/*************************************************************************/

change_PL_uPC_inst()
{
        char option;                    /* user option */

        option = uPC_inst_PL_menu();
        switch (option) {
                case 'a':
                case 'b':
                case 'c':
                case 'd':
                case 'e':
                case 'f':
                case 'g':
                case 'h':
                case 'i':
                case 'j':
                case 'k':
                case 'l':
                case 'm':
                case 'n':
                case 'o':
                case 'p':
                        uCS[state.useq.uPC].useq.inst = option - 'a';
                        break;
        };
}
```

```
/*************************************************************************/
/* change_PL_uPC_map_sel:                                                */
/*************************************************************************/

change_PL_uPC_map_sel()
{
     char option;              /* user option */

     option = uPC_map_sel_PL_menu();
     switch (option) {
          case 'a':
          case 'b':
          case 'c':
          case 'd':
               uCS[state.useq.uPC].useq.map_sel = option - 'a';
               break;
     };
}


/*************************************************************************/
/* change_PL_op_size_sel:                                                */
/*************************************************************************/

change_PL_op_size_sel()
{
     char option;              /* user option */

     option = op_size_sel_PL_menu();
     switch (option) {
          case 'a':
          case 'b':
          case 'c':
          case 'd':
               uCS[state.useq.uPC].op_size.select = option - 'a';
               break;
     };
}
```

```
/*************************************************************************/
/* change_PL_op_size:                                                    */
/*************************************************************************/

change_PL_op_size()
{
     char option;              /* user option */

     option = op_size_PL_menu();
     switch (option) {
          case 'a':
               uCS[state.useq.uPC].op_size.data = 0;
               break;
          case 'b':
               uCS[state.useq.uPC].op_size.data = 1;
               break;
          case 'c':
               uCS[state.useq.uPC].op_size.data = 2;
               break;
          };
}


/*************************************************************************/
/* change_UIDB_sel:                                                      */
/*************************************************************************/

change_UIDB_sel()
{
     char option;              /* user option */

     option = UIDB_sel_menu();
     switch (option) {
          case 'a':
          case 'b':
          case 'c':
          case 'd':
          case 'e':
          case 'f':
          case 'g':
          case 'h':
          case 'i':
          case 'j':
          case 'k':
          case 'l':
          case 'm':
               uCS[state.useq.uPC].UIDB_sel = option - 'a';
               break;
     };
}
```

```
/*************************************************************************/
/* change_uCS2:                                                          */
/*************************************************************************/

change_uCS2()
{
     int uPC;                    /* microprogram counter */
     char option;                /* user option */

     uPC = state.useq.uPC;
     do {
          option = uCS_menu2();
          switch (option) {
               case 'a':
                    change_PL_addr_data_ALU_inst("Data");
                    break;
               case 'b':
                    printf("\n\n");
                    printf("New data_ALU.OE = ");
                    scanf("%x", &uCS[uPC].data_ALU.OE);
                    break;
               case 'c':
                    printf("\n\n");
                    printf("New data_ALU.A_bus = ");
                    scanf("%x", &uCS[uPC].data_ALU.A_bus);
                    break;
               case 'd':
                    change_PL_A_B_bus_sel("Data", 'A');
                    break;
               case 'e':
                    printf("\n\n");
                    printf("New data_ALU.B_bus = ");
                    scanf("%x", &uCS[uPC].data_ALU.B_bus);
                    break;
               case 'f':
                    change_PL_A_B_bus_sel("Data", 'B');
                    break;
               case 'g':
                    change_PL_addr_data_ALU_inst("Address");
                    break;
               case 'h':
                    printf("\n\n");
                    printf("New addr_ALU.OE = ");
                    scanf("%x", &uCS[uPC].addr_ALU.OE);
                    break;
               case 'i':
                    printf("\n\n");
                    printf("New addr_ALU.A_bus = ");
                    scanf("%x", &uCS[uPC].addr_ALU.A_bus);
                    break;
               case 'j':
                    change_PL_A_B_bus_sel("Address", 'A');
                    break;
               case 'k':
                    printf("\n\n");
                    printf("New addr_ALU.B_bus = ");
                    scanf("%x", &uCS[uPC].addr_ALU.B_bus);
                    break;
```

```
                case '1':
                        change_PL_A_B_bus_sel("Address", 'B');
                        break;
                case 'm':
                        printf("\n\n");
                        printf("New MAR.load = ");
                        scanf("%x", &uCS[uPC].MAR.load);
                        break;
                case 'n':
                        printf("\n\n");
                        printf("New const_addr = ");
                        scanf("%x", &uCS[uPC].const_addr);
                        break;
                case 'o':
                        change_uCS3();
                        option = 'p';
                        break;
        };
    } while (option != 'p');
}


/*************************************************************************/
/* change_PL_A_B_bus_sel:                                                */
/*************************************************************************/

change_PL_A_B_bus_sel(ALU, bus)
char *ALU;                      /* option string */
char bus;                       /* option character */
{
    char option;                /* user option */

    option = A_B_bus_sel_PL_menu(ALU, bus);
    switch (option) {
        case 'a':
        case 'b':
        case 'c':
        case 'd':
        case 'e':
        case 'f':
        case 'g':
        case 'h':
            if ((*ALU == 'D') && (bus == 'A'))
                uCS[state.useq.uPC].data_ALU.A_bus_sel =
                                                option - 'a';
            else if ((*ALU == 'D') && (bus == 'B'))
                uCS[state.useq.uPC].data_ALU.B_bus_sel =
                                                option - 'a';
            else if ((*ALU == 'A') && (bus == 'A'))
                uCS[state.useq.uPC].addr_ALU.A_bus_sel =
                                                option - 'a';
            else if ((*ALU == 'A') && (bus == 'B'))
                uCS[state.useq.uPC].addr_ALU.B_bus_sel =
                                                option - 'a';
            break;
    };
}
```

```
/*************************************************************************/
/* change_PL_addr_data_ALU_inst:                                         */
/*************************************************************************/

change_PL_addr_data_ALU_inst(s)
char *s;                        /* option string */
{
     char option;               /* user option */

     do {
          option = addr_data_ALU_inst_PL_menu(s);
          switch (option) {
               case 'a':
                    change_PL_addr_data_ALU_src(s);
                    break;
               case 'b':
                    change_PL_addr_data_ALU_funct(s);
                    break;
               case 'c':
                    change_PL_addr_data_ALU_dest(s);
                    break;
               case 'd':
                    printf("\n\n");
                    printf("Hex instruction (Dest Funct Src) = ");
                    if (*s == 'D')
                         scanf("%x %x %x",
                               &uCS[state.useq.uPC].data_ALU.dest,
                               &uCS[state.useq.uPC].data_ALU.funct,
                               &uCS[state.useq.uPC].data_ALU.src);
                    else
                         scanf("%x %x %x",
                               &uCS[state.useq.uPC].addr_ALU.dest,
                               &uCS[state.useq.uPC].addr_ALU.funct,
                               &uCS[state.useq.uPC].addr_ALU.src);
                    break;
          };
     } while ((option != 'd') && (option != 'e'));
}
```

```
/*************************************************************************/
/* change_PL_addr_data_ALU_src:                                          */
/*************************************************************************/

change_PL_addr_data_ALU_src(s)
char *s;                        /* option string */
{
     char option;               /* user option */

     option = addr_data_ALU_src_PL_menu(s);
     switch (option) {
          case 'a':
          case 'b':
          case 'c':
          case 'd':
          case 'e':
          case 'f':
          case 'g':
          case 'h':
               if (*s == 'A')
                    uCS[state.useq.uPC].addr_ALU.src =
                                                    option - 'a';
               else
                    uCS[state.useq.uPC].data_ALU.src =
                                                    option - 'a';
               break;
     };
}


/*************************************************************************/
/* change_PL_addr_data_ALU_funct:                                        */
/*************************************************************************/

change_PL_addr_data_ALU_funct(s)
char *s;                        /* option string */
{
     char option;               /* user option */

     option = addr_data_ALU_funct_PL_menu(s);
     switch (option) {
          case 'a':
          case 'b':
          case 'c':
          case 'd':
          case 'e':
          case 'f':
          case 'g':
          case 'h':
               if (*s == 'A')
                    uCS[state.useq.uPC].addr_ALU.funct =
                                                    option - 'a';
               else
                    uCS[state.useq.uPC].data_ALU.funct =
                                                    option - 'a';
               break;
     };
}
```

```
/*************************************************************************/
/* change_PL_addr_data_ALU_dest:                                         */
/*************************************************************************/

change_PL_addr_data_ALU_dest(s)
char *s;                        /* option string */
{
     char option;               /* user option */

     option = addr_data_ALU_dest_PL_menu(s);
     switch (option) {
          case 'a':
          case 'b':
          case 'c':
          case 'd':
          case 'e':
          case 'f':
          case 'g':
          case 'h':
               if (*s == 'A')
                    uCS[state.useq.uPC].addr_ALU.dest =
                                                        option - 'a';
               else
                    uCS[state.useq.uPC].data_ALU.dest =
                                                        option - 'a';
               break;
     };
}
```

```
/*****************************************************************/
/* change_uCS3:                                                  */
/*****************************************************************/

change_uCS3()
{
     int uPC;                    /* microprogram counter */
     char option;                /* user option */

     uPC = state.useq.uPC;
     do {
          option = uCS_menu3();
          switch (option) {
               case 'a':
                    printf("\n\n");
                    printf("New DBI.load = ");
                    scanf("%x", &uCS[uPC].DBI.load);
                    break;
               case 'b':
                    printf("\n\n");
                    printf("New MM.select = ");
                    scanf("%x", &uCS[uPC].MM.select);
                    break;
               case 'c':
                    printf("\n\n");
                    printf("New MM.read = ");
                    scanf("%x", &uCS[uPC].MM.read);
                    break;
               case 'd':
                    change_PL_MM_size_sel();
                    break;
               case 'e':
                    change_PL_MM_size();
                    break;
               case 'f':
                    printf("\n\n");
                    printf("New TCC.X_load = ");
                    scanf("%x", &uCS[uPC].TCC.X_load);
                    break;
               case 'g':
                    printf("\n\n");
                    printf("New TCC.N_load = ");
                    scanf("%x", &uCS[uPC].TCC.N_load);
                    break;
               case 'h':
                    printf("\n\n");
                    printf("New TCC.Z_load = ");
                    scanf("%x", &uCS[uPC].TCC.Z_load);
                    break;
               case 'i':
                    printf("\n\n");
                    printf("New TCC.V_load = ");
                    scanf("%x", &uCS[uPC].TCC.V_load);
                    break;
               case 'j':
                    printf("\n\n");
                    printf("New TCC.C_load = ");
                    scanf("%x", &uCS[uPC].TCC.C_load);
                    break;
```

```
                case 'k':
                        change_PL_SR_sel();
                        break;
                case 'l':
                        printf("\n\n");
                        printf("New Cin.data = ");
                        scanf("%x", &uCS[uPC].Cin.data);
                        break;
                case 'm':
                        change_PL_Cin_sel();
                        break;
                case 'n':
                        change_uCS4();
                        option = 'o';
                        break;
        };
    } while (option != 'o');
}


/*************************************************************************/
/* change_PL_MM_size_sel:                                                */
/*************************************************************************/

change_PL_MM_size_sel()
{
    char option;            /* user option */

    option = MM_size_sel_PL_menu();
    switch (option) {
        case 'a':
        case 'b':
        case 'c':
        case 'd':
            uCS[state.useq.uPC].MM.size_sel = option - 'a';
            break;
    };
}


/*************************************************************************/
/* change_PL_MM_size:                                                    */
/*************************************************************************/

change_PL_MM_size()
{
    char option;            /* user option */

    option = MM_size_PL_menu();
    switch (option) {
        case 'a':
        case 'b':
        case 'c':
            uCS[state.useq.uPC].MM.size = option - 'a';
            break;
    };
}
```

```
/*****************************************************************************/
/* change_PL_SR_sel:                                                         */
/*****************************************************************************/

change_PL_SR_sel()
{
     char option;              /* user option */

     option = SR_sel_PL_menu();
     switch (option) {
          case 'a':
          case 'b':
          case 'c':
          case 'd':
               uCS[state.useq.uPC].SR_sel = option - 'a';
               break;
     };
}


/*****************************************************************************/
/* change_PL_Cin_sel:                                                        */
/*****************************************************************************/

change_PL_Cin_sel()
{
     char option;              /* user option */

     option = Cin_sel_PL_menu();
     switch (option) {
          case 'a':
          case 'b':
               uCS[state.useq.uPC].Cin.select = option - 'a';
               break;
     };
}
```

```
/*************************************************************************/
/* change_uCS4:                                                          */
/*************************************************************************/

change_uCS4()
{
     int uPC;                    /* microprogram counter */
     char option;                /* user option */

     uPC = state.useq.uPC;
     do {
          option = uCS_menu4();
          switch (option) {
               case 'a':
                    printf("\n\n");
                    printf("New mult.data_1_load = ");
                    scanf("%x", &uCS[uPC].mult.data_1_load);
                    break;
               case 'b':
                    printf("\n\n");
                    printf("New mult.data_2_load = ");
                    scanf("%x", &uCS[uPC].mult.data_2_load);
                    break;
               case 'c':
                    change_PL_shifter_inst();
                    break;
               case 'd':
                    printf("\n\n");
                    printf("New shifter.shift_load = ");
                    scanf("%x", &uCS[uPC].shifter.shift_load);
                    break;
               case 'e':
                    printf("\n\n");
                    printf("New shifter.data_load = ");
                    scanf("%x", &uCS[uPC].shifter.data_load);
                    break;
               case 'f':
                    printf("\n\n");
                    printf("New shifter.CX_load = ");
                    scanf("%x", &uCS[uPC].shifter.CX_load);
                    break;
               case 'g':
                    printf("\n\n");
                    printf("New feedback.load = ");
                    scanf("%x", &uCS[uPC].feedback.load);
                    break;
               case 'h':
                    change_PL_feedback_size_sel();
                    break;
               case 'i':
                    change_PL_feedback_size();
                    break;
               case 'j':
                    printf("\n\n");
                    printf("New feedback.scale_sel = ");
                    scanf("%x", &uCS[uPC].feedback.scale_sel);
                    break;
```

```
                case 'k':
                        printf("\n\n");
                        printf("New EAR.load = ");
                        scanf("%x", &uCS[uPC].EAR.load);
                        break;
                case 'l':
                        printf("\n\n");
                        printf("New EAR.select = ");
                        scanf("%x", &uCS[uPC].EAR.select);
                        break;
                case 'm':
                        printf("\n\n");
                        printf("New data_ALU_mask = ");
                        scanf("%x", &uCS[uPC].data_ALU_mask);
                        break;
                case 'n':
                        change_uCS1();
                        option = 'o';
                        break;
        };
    } while (option != 'o');
}


/*************************************************************************/
/* change_PL_shifter_inst:                                               */
/*************************************************************************/

change_PL_shifter_inst()
{
    char option;            /* user option */

    option = shifter_inst_PL_menu();
    switch (option) {
        case 'a':
        case 'b':
        case 'c':
        case 'd':
        case 'e':
        case 'f':
        case 'g':
        case 'h':
            uCS[state.useq.uPC].shifter.inst = option - 'a';
            break;
    };
}
```

```
/*****************************************************************************/
/* change_PL_feedback_size_sel:                                              */
/*****************************************************************************/

change_PL_feedback_size_sel()
{
     char option;              /* user option */

     option = feedback_size_sel_PL_menu();
     switch (option) {
          case 'a':
          case 'b':
          case 'c':
               uCS[state.useq.uPC].feedback.size_sel = option - 'a';
               break;
     };
}


/*****************************************************************************/
/* change_PL_feedback_size:                                                  */
/*****************************************************************************/

change_PL_feedback_size()
{
     char option;              /* user option */

     option = feedback_size_PL_menu();
     switch (option) {
          case 'a':
          case 'b':
          case 'c':
               uCS[state.useq.uPC].feedback.size = option - 'a';
               break;
     };
}
```

```
/*************************************************************************/
/* edit_uCS:                                                             */
/*************************************************************************/

edit_uCS()
{
     char option;

     do {

          printf("\n\n\n\n\n");
          hex(state.useq.uPC, 3);
          printf("Pipeline bits for uPC = %s\n\n", hex_string);
          print_PL(uCS[state.useq.uPC]);

          printf("\n");
          printf("I=inc uPC  D=dec uPC  N=new uPC  C=clear  ");
          printf("E=edit uCS  S=save  X=exit\n");
          option = tolower(getchar());
          while (getchar() != '\n')
               ;

          switch (option) {
               case 'i':
                    if (state.useq.uPC < 4095)
                         state.useq.uPC++;
                    break;
               case 'd':
                    if (state.useq.uPC > 0)
                         --state.useq.uPC;
                    break;
               case 'c':
                    uCS[state.useq.uPC].useq.NA = 0;
                    uCS[state.useq.uPC].useq.inst = 0;
                    uCS[state.useq.uPC].useq.map_sel = 0;
                    uCS[state.useq.uPC].useq.CC_logic_level = 0;
                    uCS[state.useq.uPC].useq.CC_sel = 0;
                    uCS[state.useq.uPC].IR_load = 0;
                    uCS[state.useq.uPC].EWR_load = 0;
                    uCS[state.useq.uPC].op_size.load = 0;
                    uCS[state.useq.uPC].op_size.select = 0;
                    uCS[state.useq.uPC].op_size.data = 0;
                    uCS[state.useq.uPC].UIDB_sel = 0;
                    uCS[state.useq.uPC].data_ALU.src = 0;
                    uCS[state.useq.uPC].data_ALU.funct = 0;
                    uCS[state.useq.uPC].data_ALU.dest = 0;
                    uCS[state.useq.uPC].data_ALU.OE = 0;
                    uCS[state.useq.uPC].data_ALU.A_bus = 0;
                    uCS[state.useq.uPC].data_ALU.A_bus_sel = 0;
                    uCS[state.useq.uPC].data_ALU.B_bus = 0;
                    uCS[state.useq.uPC].data_ALU.B_bus_sel = 0;
                    uCS[state.useq.uPC].addr_ALU.src = 0;
                    uCS[state.useq.uPC].addr_ALU.funct = 0;
                    uCS[state.useq.uPC].addr_ALU.dest = 0;
                    uCS[state.useq.uPC].addr_ALU.OE = 0;
                    uCS[state.useq.uPC].addr_ALU.A_bus = 0;
                    uCS[state.useq.uPC].addr_ALU.A_bus_sel = 0;
                    uCS[state.useq.uPC].addr_ALU.B_bus = 0;
                    uCS[state.useq.uPC].addr_ALU.B_bus_sel = 0;
```

```
                    uCS[state.useq.uPC].MAR.load = 0;
                    uCS[state.useq.uPC].const_addr = 0;
                    uCS[state.useq.uPC].DBI.load = 0;
                    uCS[state.useq.uPC].MM.select = 0;
                    uCS[state.useq.uPC].MM.read = 0;
                    uCS[state.useq.uPC].MM.size_sel = 0;
                    uCS[state.useq.uPC].MM.size = 0;
                    uCS[state.useq.uPC].TCC.X_load = 0;
                    uCS[state.useq.uPC].TCC.N_load = 0;
                    uCS[state.useq.uPC].TCC.Z_load = 0;
                    uCS[state.useq.uPC].TCC.V_load = 0;
                    uCS[state.useq.uPC].TCC.C_load = 0;
                    uCS[state.useq.uPC].SR_sel = 0;
                    uCS[state.useq.uPC].Cin.data = 0;
                    uCS[state.useq.uPC].Cin.select = 0;
                    uCS[state.useq.uPC].mult.data_1_load = 0;
                    uCS[state.useq.uPC].mult.data_2_load = 0;
                    uCS[state.useq.uPC].shifter.inst = 0;
                    uCS[state.useq.uPC].shifter.shift_load = 0;
                    uCS[state.useq.uPC].shifter.data_load = 0;
                    uCS[state.useq.uPC].shifter.CX_load = 0;
                    uCS[state.useq.uPC].feedback.load = 0;
                    uCS[state.useq.uPC].feedback.size_sel = 0;
                    uCS[state.useq.uPC].feedback.size = 0;
                    uCS[state.useq.uPC].feedback.scale_sel = 0;
                    uCS[state.useq.uPC].EAR.load = 0;
                    uCS[state.useq.uPC].EAR.select = 0;
                    uCS[state.useq.uPC].data_ALU_mask = 0;
                    break;
               case 'n':
                    do {
                         printf("New uPC = ");
                         scanf("%x", &state.useq.uPC);
                         while (getchar() != '\n')
                              ;
                    } while((state.useq.uPC < 0) ||
                            (state.useq.uPC > 4095));
                    break;
               case 'e':
                    change_uCS1();
                    break;
               case 's':
                    save_uCS();
                    break;
          };
     } while (option != 'x');
}
```

```
/*************************************************************************/
/* edit_MM:                                                              */
/*************************************************************************/

edit_MM()
{
     int addr = 0;
     int i;
     int j;
     int temp;
     int value;
     int inc = 2;
     char MM_string[100];
     char addr_string[100];
     char option;

     do {

          printf("\n\n\n\n\n");
          for (i = addr-10*inc; i < addr+10*inc; i = i + inc)
               if ((i < 0) || (i > 65535))
                    printf("\n");
               else {
                    hex(i,4);
                    strcpy(addr_string, hex_string);
                    MM_string[0] = ' ';
                    for (j = 0; j < inc; j++) {
                         hex(MM[i + j], 2);
                         strcat(MM_string, hex_string);
                    };
                    printf("                              %s  %s\n",
                           addr_string, MM_string);
               };
          printf("\n");
          printf("S=Save   I=inc   D=dec   N=new address");
          printf("   E=Edit MM   L=Length   X=exit\n");
          option = tolower(getchar());
          while (getchar() != '\n')
               ;
```

```
            switch (option) {
                case 's':
                    save_MM();
                    break;
                case 'l':
                    printf("\n");
                    printf("Enter the number of bytes per line ==> ");
                    scanf("%x", &inc);
                    while (getchar() != '\n')
                        ;
                    break;
                case 'i':
                    if (addr+10*inc < 65535)
                        addr = addr + inc*10;
                    break;
                case 'd':
                    if (addr-10*inc > 0)
                        addr = addr - 10*inc;
                    break;
                case 'n':
                    do {
                        printf("New address = ");
                        scanf("%x", &addr);
                        while (getchar() != '\n')
                            ;
                    } while((addr < 0) ||
                            (addr > 65535));
                    break;
                case 'e':
                    do {
                        printf("Edit address = ");
                        scanf("%x", &temp);
                    } while((temp < 0) || (temp > 65535));
                    printf("\n\n");
                    printf("New value    = ");
                    scanf("%x", &value);
                    while (getchar() != '\n')
                        ;
                    for (j = inc - 1; j >= 0; --j) {
                        MM[temp + j] = value & 0x000000ff;
                        value = value >> 8;
                    };
                    break;
            };
        } while (option != 'x');
}
```

```
/*****************************************************************************/
/* execute.c                                                                 */
/*                                                                           */
/*      Simulates the execution of N microinstructions, N target             */
/* instructions, or 1 target program and prints the current state            */
/* of the host and the current state of the target machine.                  */
/*****************************************************************************/
```

```
/*************************************************************************/
/* execute_1_uinst:                                                      */
/*      Executes 1 microinstruction that is pointed to by the uPC        */
/* and then prints the state of the host.                                */
/*************************************************************************/

execute_1_uinst()
{
     execute_uinst();
     if (trace == TRUE) {
          printf("\n");
          printf("Press <RETURN> to continue. ");
          while (getchar() != '\n')
               ;
     };
     print_host_state();
}


/*************************************************************************/
/* execute_N_uinst:                                                      */
/*      Executes microinstructions beginning with the one pointed to     */
/* by the uPC.  Execution continues until a breakpoint is reached,       */
/* an error occurs (an illegal microinstruction), or until N             */
/* microinstructions have been executed.  Then the state of the          */
/* the host is printed.                                                  */
/*************************************************************************/

execute_N_uinst()
{
     int n;                          /* number of microinstructions */

     printf("\n\n");
     printf("Number of microinstructions  ");
     scanf("%d", &n);
     while (getchar() != '\n')
          ;

     do {
          execute_uinst();
          --n;
     } while ((n >= 0) && (state.useq.uPC != breakpoint) &&
               (error != TRUE));

     if (trace == TRUE) {
          printf("\n");
          printf("Press <RETURN> to continue. ");
          while (getchar() != '\n')
               ;
     };
     print_host_state();
}
```

```
/*************************************************************************/
/* execute_1_inst:                                                       */
/*      Executes 1 target level instruction which is pointed to by       */
/* the PC.  Execution continues until the IR PROM is enabled, an         */
/* error occurs (an illegal microinstruction), or a breakpoint is        */
/* reached.  Then the state of the target machine is printed.            */
/*************************************************************************/

execute_1_inst()
{
     inst_cnt = 0;                  /* initialize trace counter */
     do {
          execute_uinst();
     } while ((state.PL.useq.map_sel != 2) &&
               (state.useq.uPC != breakpoint) && (error != TRUE));

     if (trace == TRUE) {
          printf("\n");
          printf("Press <RETURN> to continue. ");
          while (getchar() != '\n')
               ;
     };
     print_target_state();
}
```

```
/*************************************************************************/
/* execute_N_inst:                                                       */
/*      Executes N target level instructions, beginning with the one     */
/* pointed to by the PC.  Execution continues until a breakpoint is      */
/* reached, an error occurs (an illegal microinstruction), or N target   */
/* level instructions have been executed.  Then the state of the         */
/* target machine is printed.                                            */
/*************************************************************************/

execute_N_inst()
{
     int n;                              /* number of instructions */

     printf("\n\n");
     printf("Number of target instructions  ");
     scanf("%d", &n);
     while (getchar() != '\n')
          ;

     do {

          inst_cnt = 0;                  /* initialize instruction counter */
          do {
               execute_uinst();
          } while ((state.PL.useq.map_sel != 2) &&
                   (state.useq.uPC != breakpoint) && (error != TRUE));
          --n;
     } while ((n >= 0) && (state.useq.uPC != breakpoint) &&
              (error != TRUE));


     if (trace == TRUE) {
          printf("\n");
          printf("Press <RETURN> to continue. ");
          while (getchar() != '\n')
               ;
     };
     print_target_state();
}
```

```
/*************************************************************************/
/* execute_prog:                                                         */
/*      Executes target level instructions, beginning with the one       */
/* pointed to by the PC.  Execution continues until a breakpoint is      */
/* reached, an error occurs (an illegal microinstruction), or until      */
/* the STOP instruction is executed.  Then the state of the target       */
/* machine is printed.                                                   */
/*************************************************************************/

execute_prog()
{
     prog_cnt = 0;                  /* initialize program counter */

     do {

          inst_cnt = 0;             /* initialize instruction counter */
          do {
               execute_uinst();
          } while ((state.PL.useq.map_sel != 2) &&
                   (state.useq.uPC != breakpoint) && (error != TRUE));

     } while ((state.useq.uPC != breakpoint) && (error != TRUE));

     if (trace == TRUE) {
          printf("\n");
          printf("Press <RETURN> to continue. ");
          while (getchar() != '\n')
               ;
     };
     print_target_state();
}
```

```
/*************************************************************************/
/* execute_uinst:                                                        */
/*      Executes 1 microinstruction.  First it loads the PL with the     */
/* microinstruction pointed to by the uPC, and then checks for an        */
/* illegal microinstruction.  If there are no errors, execution          */
/* continues.  If the trace mode is enabled, the current uPC is          */
/* printed.  Then the state variables are set.                           */
/*************************************************************************/

execute_uinst()
{
     state.PL = uCS[state.useq.uPC];
     check_error();
     if (error == FALSE) {
          if (trace == 1) {
               prog_cnt++;
               inst_cnt++;
               if ((inst_cnt % 20) == 0) {
                    printf("Prog Cycles = %d   Inst Cycles = %d   ",
                           prog_cnt, inst_cnt);
                    printf("uPC = %x\n", state.useq.uPC);
                    printf("\n");
                    printf("Press <RETURN> to continue. ");
                    while (getchar() != '\n')
                         ;
               }
               else
                    printf("Prog Cycles = %d   Inst Cycles = %d   ",
                           prog_cnt, inst_cnt);
                    printf("uPC = %x\n", state.useq.uPC);
          };

          /* set values onto all busses */
          state.useq.CC = useq_CC();
          set_UIDB();
          set_A_B_bus();
          data_addr_ALU_and_set_LIDB();
          set_MM_busses();
          set_useq_bus();
```

```
/* load latches */
state.LIDB = bus.LIDB;
state.HCC.C = data_CC.C;
state.HCC.Z = data_CC.Z;
state.HCC.V = data_CC.V;
state.HCC.N = data_CC.N;
switch (state.PL.SR_sel) {
     case 0:
          if (state.PL.TCC.C_load == 1)
               state.TCC.C = data_CC.C;
          if (state.PL.TCC.V_load == 1)
               state.TCC.V = data_CC.V;
          if (state.PL.TCC.Z_load == 1)
               state.TCC.Z = data_CC.Z;
               if (state.PL.TCC.N_load == 1)
               state.TCC.N = data_CC.N;
          if (state.PL.TCC.X_load == 1)
               state.TCC.X = data_CC.X;
          break;
     case 1:
          if (state.PL.TCC.C_load == 1)
               state.TCC.C = bus.UIDB & 0x00000001;
          if (state.PL.TCC.V_load == 1)
               state.TCC.V = (bus.UIDB & 0x00000002) >> 1;
          if (state.PL.TCC.Z_load == 1)
               state.TCC.Z = (bus.UIDB & 0x00000004) >> 2;
          if (state.PL.TCC.N_load == 1)
               state.TCC.N = (bus.UIDB & 0x00000008) >> 3;
          if (state.PL.TCC.X_load == 1)
               state.TCC.X = (bus.UIDB & 0x00000010) >> 4;
          break;
     case 2:
          if (state.PL.TCC.C_load == 1)
               state.TCC.C = state.shifter.C;
          if (state.PL.TCC.V_load == 1)
               state.TCC.V = state.shifter.V;
          if (state.PL.TCC.Z_load == 1)
               state.TCC.Z = state.shifter.Z;
          if (state.PL.TCC.N_load == 1)
               state.TCC.N = state.shifter.N;
          if (state.PL.TCC.X_load == 1)
               state.TCC.X = state.shifter.X;
          break;
     case 3:
          if (state.PL.TCC.C_load == 1)
               state.TCC.C = addr_CC.C;
          if (state.PL.TCC.V_load == 1)
               state.TCC.V = addr_CC.V;
          if (state.PL.TCC.Z_load == 1)
               state.TCC.Z = addr_CC.Z;
          if (state.PL.TCC.N_load == 1)
               state.TCC.N = addr_CC.N;
          if (state.PL.TCC.X_load == 1)
               state.TCC.X = addr_CC.X;
          break;
};
```

```
        if (state.PL.MAR.load == 1)
             state.MAR = bus.LIDB;
        if (state.PL.IR_load == 1)
             state.PIR = bus.MM_data & 0x0000ffff;
        if (state.PL.EWR_load == 1)
             state.EWR = bus.MM_data & 0x0000ffff;
        if (state.PL.DBI.load == 1)
             state.DBI = bus.UIDB;
        if ((state.PL.MM.select & state.PL.MM.read
           & ~state.PL.IR_load & ~state.PL.EWR_load) == 1)
             state.DBI = bus.MM_data;
        if (state.PL.mult.data_1_load == 1)
             state.mult.data_1 = bus.LIDB;
        if (state.PL.mult.data_2_load == 1)
             state.mult.data_2 = bus.LIDB;
        if (state.PL.shifter.data_load == 1)
             state.shifter.data = bus.LIDB;
        if (state.PL.shifter.shift_load == 1)
             state.shifter.shift = bus.LIDB & 0x3F;
        if (state.PL.shifter.CX_load == 1) {
             state.shifter.C = state.TCC.C;
             state.shifter.X = state.TCC.X;
        }
        if (state.PL.feedback.load == 1)
             state.feedback = bus.LIDB;
        if (state.PL.EAR.load == 1) {
             if (state.PL.EAR.select == 0)
                  state.EAR = state.IR & 0x003f;
             else
                  state.EAR = ((state.IR >> 9) & 0x07)
                            | ((state.IR >> 3) & 0x038);
        };

        /* simulate functional units */
        main_memory();
        op_size();
        useq();
    };
}
```

```
/*************************************************************************/
/* check_error:                                                          */
/*      Checks for an illegal microinstruction.  If there is an error,   */
/* it is printed, and the error flag is set to TRUE, thus halting        */
/* execution.                                                            */
/*************************************************************************/

check_error()
{
     error = FALSE;                /* initialize error flag */

     /* Check if more than one device is trying to output */
     /* data onto the LIDB */
     if (state.PL.addr_ALU.OE + state.PL.data_ALU.OE > 1) {
          printf("\n\n");
          printf("ERROR:  LIDB bus contention.");
          error = TRUE;
     };

     /* check for more than one source on the main memory data bus */
     if ((state.PL.MM.select & ~state.PL.MM.read)
        + state.PL.MM.read > 1) {
          printf("\n\n");
          printf("ERROR:  Main memory data bus contention.");
          error = TRUE;
     };

     /* check for two sources for the DBI */
     if ((state.PL.MM.select & state.PL.MM.read
        & ~state.PL.IR_load & ~state.PL.EWR_load)
        + state.PL.DBI.load > 1) {
          printf("\n\n");
          printf("ERROR:  DBI contention.");
          error = TRUE;
     };

     /* Check for an illegal microsequencer instruction */
     /* (i.e. - one that is not yet implemented) */
     switch (state.PL.useq.inst) {
          case JZ:
          case JSRP:
          case CJV:
          case JRP:
          case RFCT:
          case RPCT:
          case CJPP:
          case LDCT:
          case LOOP:
               printf("\n\n");
               printf("ERROR:  Illegal microsequencer instruction");
               error = TRUE;
               break;
     }
```

```
        if (state.useq.uPC == 6) {
              printf("\n\n");
              printf("ERROR:  STOP instruction reached");
              error = TRUE;
        }

        if (state.useq.uPC == 7) {
              printf("\n\n");
              printf("TRAP instruction reached");
              error = TRUE;
        }

        if (error == TRUE) {
              printf("\n\n");
              printf("Execution terminated.");
              while (getchar() != '\n')
                    ;
        };
}
```

```
/*************************************************************************/
/* set_UIDB:                                                             */
/*      Sets the Upper Internal Data Bus (UIDB) value if the constants   */
/* PROM, the Data Bus Interface (DBI), the Status Register (SR), the     */
/* multiplier, the barrel shifter, or the feedback latch have their      */
/* output active.                                                        */
/*************************************************************************/

set_UIDB()
{
     int R_upper;                /* upper multiplier result */
     int R_lower;                /* lower multiplier result */

     switch (state.PL.UIDB_sel) {
          case 0:
               bus.UIDB = 0;
               break;
          case 1:
               bus.UIDB = state.DBI;
               break;
          case 2:
               bus.UIDB = const[state.PL.const_addr];
               break;
          case 3:
               switch (state.op_size) {
                    case 0:
                         bus.UIDB = 1;
                         break;
                    case 1:
                         bus.UIDB = 2;
                         break;
                    case 2:
                         bus.UIDB = 4;
                         break;
               }
               break;
          case 4:
               bus.UIDB = (state.TCC.X * 16) + (state.TCC.N * 8) +
                          (state.TCC.Z * 4) + (state.TCC.V * 2) +
                          (state.TCC.C);
               break;
          case 5:
               bus.UIDB = ((state.EWR & 0xff) |
                           (((state.EWR >> 7) & 0x01) * 0xffffff00));
               break;
          case 6:
               bus.UIDB = feedback();
               break;
          case 7:
               bus.UIDB = shifter();
               break;
          case 8:
               mult(&R_upper, &R_lower);
               bus.UIDB = R_lower;
               break;
          case 9:
               mult(&R_upper, &R_lower);
               bus.UIDB = R_upper;
               break;
```

```
            case 10:
                bus.UIDB = ((state.EWR & 0xffff) |
                            (((state.EWR >> 15) & 0x01) * 0xffff0000));
                break;
            case 11:
                bus.UIDB = (state.IR >> 9) & 0x07;
                        if (bus.UIDB == 0)
                            bus.UIDB = 8;
                break;
            case 12:
                bus.UIDB = ((state.IR & 0xff) |
                            (((state.IR >> 7) & 0x01) * 0xffffff00));
                break;
        }
}
```

```
/*****************************************************************************/
/* mult:                                                                     */
/*      simulator for 32X32  hardware multiplier                             */
/*****************************************************************************/

mult(r2, r1)
unsigned int *r2;
unsigned int *r1;
{
     int i;                              /* mult loop counting variable */
     unsigned int d1;                    /* operand 1 */
     unsigned int d2_lower;              /* lower word of operand 2 */
     unsigned int d2_upper;              /* upper word of operand 2 */
     unsigned int r2_lower;              /* lower word of upper result */
     unsigned int r2_upper;              /* upper word of upper result */
     int carry;                          /* carry after 16 bit addition */

     *r1 = 0;
     *r2 = 0;
     r2_lower = 0;
     r2_upper = 0;
     d1 = state.mult.data_1;
     d2_lower = state.mult.data_2 & 0x0000ffff;
     d2_upper = (state.mult.data_2 & 0xffff0000) >> 16;

     for (i=0;i<32;i++) {
          if ((d1 & 0x00000001) == 1) {
               r2_lower = *r2 & 0x0000ffff;
               r2_upper = (*r2 & 0xffff0000) >> 16;
               r2_lower = r2_lower + d2_lower;
               carry = r2_lower >> 16;
               r2_upper = r2_upper + d2_upper + carry;
               carry = r2_upper >> 16;
               *r2 = ((r2_upper & 0x0000ffff) << 16) +
                      (r2_lower & 0x0000ffff);
          }
          else
               carry = 0;
          if ((*r2 & 0x00000001) == 1)
               *r1 = (*r1 >> 1) + (1 << 31);
          else
               *r1 = (*r1 >> 1);
          *r2 = (*r2 >> 1) + (carry << 31);
          d1 = d1 >> 1;
     };
}
```

```
/*************************************************************************/
/* feedback:                                                             */
/*      Returns the result of the feedback latch                         */
/*************************************************************************/

feedback()
{
     int size;                  /* size of feedback sign extension */
     int sign;                  /* sign of feedback size */
     int result;                /* output of feedback latch */

     result = state.feedback;

     /* perform sign extension */
     switch (state.PL.feedback.size_sel) {
          case 0:
               size = state.PL.feedback.size;
               break;
          case 1:
               size = state.op_size;
               break;
          case 2:
               size = ((state.EWR >> 11) & 0x01) + 1;
               break;
     }
     switch (size) {
          case 0:
               sign = ((result >> 7) & 0x01);
               switch (sign) {
                    case 0:
                         result = result & 0x000000ff;
                         break;
                    case 1:
                         result = result | 0xffffff00;
                         break;
               }
               break;
          case 1:
               sign = ((result >> 15) & 0x01);
               switch (sign) {
                    case 0:
                         result = result & 0x0000ffff;
                         break;
                    case 1:
                         result = result | 0xffff0000;
                         break;
               }
               break;
          case 2:
               break;
     }

     /* perform shifting if enabled */
     if (state.PL.feedback.scale_sel == 1)
          result = result << ((state.EWR >> 9) & 0x03);
     return(result);
}
```

```
/*****************************************************************************/
/* Include SHIFTER:                                                          */
/*****************************************************************************/

#include "shifter.c"


/*****************************************************************************/
/* set_A_B_bus:                                                              */
/*      Sets the A bus and the B bus values depending on the PL.             */
/*****************************************************************************/

set_A_B_bus()
{
     switch(state.PL.data_ALU.A_bus_sel) {
          case 0:
               bus.data_ALU_A = state.PL.data_ALU.A_bus;
               break;
          case 1:
               bus.data_ALU_A = state.IR & 7;
               break;
          case 2:
               bus.data_ALU_A = (state.IR >> 6) & 7;
               break;
          case 3:
               bus.data_ALU_A = (state.IR >> 9) & 7;
               break;
          case 4:
               bus.data_ALU_A = (state.IR >> 12) & 7;
               break;
          case 5:
               bus.data_ALU_A = (state.EWR >> 12) & 7;
               break;
          case 6:
               bus.data_ALU_A = state.EAR & 7;
               break;
          case 7:
               bus.data_ALU_A = state.EWR & 7;
               break;
     };
```

```
switch(state.PL.data_ALU.B_bus_sel) {
     case 0:
          bus.data_ALU_B = state.PL.data_ALU.B_bus;
          break;
     case 1:
          bus.data_ALU_B = state.IR & 7;
          break;
     case 2:
          bus.data_ALU_B = (state.IR >> 6) & 7;
          break;
     case 3:
          bus.data_ALU_B = (state.IR >> 9) & 7;
          break;
     case 4:
          bus.data_ALU_B = (state.IR >> 12) & 7;
          break;
     case 5:
          bus.data_ALU_B = (state.EWR >> 12) & 7;
          break;
     case 6:
          bus.data_ALU_B = state.EAR & 7;
          break;
     case 7:
          bus.data_ALU_B = state.EWR & 7;
          break;
};

switch(state.PL.addr_ALU.A_bus_sel) {
     case 0:
          bus.addr_ALU_A = state.PL.addr_ALU.A_bus;
          break;
     case 1:
          bus.addr_ALU_A = state.IR & 7;
          break;
     case 2:
          bus.addr_ALU_A = (state.IR >> 6) & 7;
          break;
     case 3:
          bus.addr_ALU_A = (state.IR >> 9) & 7;
          break;
     case 4:
          bus.addr_ALU_A = (state.IR >> 12) & 7;
          break;
     case 5:
          bus.addr_ALU_A = (state.EWR >> 12) & 7;
          break;
     case 6:
          bus.addr_ALU_A = state.EAR & 7;
          break;
     case 7:
          bus.data_ALU_A = state.EWR & 7;
          break;
};
```

```
        switch(state.PL.addr_ALU.B_bus_sel) {
             case 0:
                  bus.addr_ALU_B = state.PL.addr_ALU.B_bus;
                  break;
             case 1:
                  bus.addr_ALU_B = state.IR & 7;
                  break;
             case 2:
                  bus.addr_ALU_B = (state.IR >> 6) & 7;
                  break;
             case 3:
                  bus.addr_ALU_B = (state.IR >> 9) & 7;
                  break;
             case 4:
                  bus.addr_ALU_B = (state.IR >> 12) & 7;
                  break;
             case 5:
                  bus.addr_ALU_B = (state.EWR >> 12) & 7;
                  break;
             case 6:
                  bus.addr_ALU_B = state.EAR & 7;
                  break;
             case 7:
                  bus.data_ALU_A = state.EWR & 7;
                  break;
        };
}


/*************************************************************************/
/* data_addr_ALU_and_set_LIDB:                                           */
/*************************************************************************/

data_addr_ALU_and_set_LIDB()
{
        unsigned int data_F;        /* F output of data ALU) */
        unsigned int addr_F;        /* F output of address ALU */
        unsigned int temp;          /* temporary variable */
        int RAM_0;                  /* RAM_0 output */
        int Q_0;                    /* Data ALU Q_0 output */

        AM2901(state.data_ALU, bus.data_ALU_A, bus.data_ALU_B,
               state.PL.data_ALU, &data_F, TRUE);
        AM2901(state.addr_ALU, bus.addr_ALU_A, bus.addr_ALU_B,
               state.PL.addr_ALU, &addr_F, FALSE);
```

```
switch(state.PL.data_ALU.dest) {
     case QREG:
          dest_size(&state.data_ALU.Q,&data_F);
          break;
     case NOP:
          break;
     case RAMA:
          dest_size(&state.data_ALU.RAM[bus.data_ALU_B], &data_F);
          break;
     case RAMF:
          dest_size(&state.data_ALU.RAM[bus.data_ALU_B], &data_F);
          break;
     case RAMQD:
          RAM_0 = data_F & 0x01;
          temp = data_F >> 1;
          dest_size(&state.data_ALU.RAM[bus.data_ALU_B], &temp);
          Q_0 = state.data_ALU.Q & 0x01;
          temp = (state.data_ALU.Q >> 1) | (RAM_0 << 31);
          dest_size(&state.data_ALU.Q, &temp);
          break;
     case RAMD:
          temp = data_F >> 1;
          dest_size(&state.data_ALU.RAM[bus.data_ALU_B], &temp);
          break;
     case RAMQU:
          temp = (data_F << 1) |
                 ((state.data_ALU.Q >> 31) & 0x01);
          dest_size(&state.data_ALU.RAM[bus.data_ALU_B], &temp);
          temp = state.data_ALU.Q << 1;
          dest_size(&state.data_ALU.Q, &temp);
          if (state.PL.addr_ALU.dest == RAMQU)
               state.data_ALU.Q = state.data_ALU.Q |
                                  ((state.addr_ALU.Q >> 31) & 0x01);
          break;
     case RAMU:
          temp = data_F << 1;
          dest_size(&state.data_ALU.RAM[bus.data_ALU_B], &temp);
          break;
};
```

```
        switch(state.PL.addr_ALU.dest) {
             case QREG:
                  state.addr_ALU.Q = addr_F;
                  break;
             case NOP:
                  break;
             case RAMA:
                  state.addr_ALU.RAM[bus.addr_ALU_B] = addr_F;
                  break;
             case RAMF:
                  state.addr_ALU.RAM[bus.addr_ALU_B] = addr_F;
                  break;
             case RAMQD:
                  state.addr_ALU.RAM[bus.addr_ALU_B] = addr_F >> 1;
                  state.addr_ALU.Q = state.addr_ALU.Q >> 1;
                  if (state.PL.data_ALU.dest == RAMQD) {
                       state.addr_ALU.Q = state.addr_ALU.Q | (Q_0 << 31);
                  };
                  break;
             case RAMD:
                  state.addr_ALU.RAM[bus.addr_ALU_B] = addr_F >> 1;
                  break;
             case RAMQU:
                  state.addr_ALU.RAM[bus.addr_ALU_B] = (addr_F << 1);
                  state.addr_ALU.Q = (state.addr_ALU.Q << 1)
                                   | ((~state.HCC.N) & 0x01);
                  break;
             case RAMU:
                  state.addr_ALU.RAM[bus.addr_ALU_B] = addr_F << 1;
                  break;
        };
}
```

```
/*************************************************************************/
/* AM2901:                                                               */
/*     Sets the Cin, finds the source operands corresponding to the      */
/* source field, performs the function corresponding to the function     */
/* field, and sets the pointer to the result.  Also, the flags are set   */
/* if the load lines are active and the ALU is the data ALU.             */
/*************************************************************************/

AM2901(ALU_state, A_bus, B_bus, ALU_PL, F, set_CC)
struct ALU_state_type ALU_state;     /* ALU type */
int A_bus;                           /* ALU A bus value */
int B_bus;                           /* ALU B bus value */
struct ALU_PL_type ALU_PL;           /* ALU instruction */
int *F;                              /* pointer to output */
int set_CC;                          /* 1 = data ALU */
                                     /* 0 = address ALU */

{
     unsigned int R;           /* ALU mux port R */
     unsigned int S;           /* ALU mux port S */
     int C;                    /* temporary C flag */
     int V;                    /* temporary V flag */
     int Z;                    /* temporary Z flag */
     int N;                    /* temporary N flag */
     int X;                    /* temporary X flag */
     int Cin;                  /* temporary Cin */

     if (!set_CC) {
          if ((state.PL.addr_ALU.funct == 0) ||
              (state.PL.addr_ALU.funct == 4))
               Cin = 0;
          else
               Cin = 1;
     }
     else {
          if (state.PL.Cin.select == 1)
               Cin = state.TCC.X;
          else
               Cin = state.PL.Cin.data;
     }
```

```
/* Set the source operands according to the source bits */
switch(ALU_PL.src) {
     case AQ:
          R = size(ALU_state.RAM[A_bus], set_CC);
          S = size(ALU_state.Q, set_CC);
          break;
     case AB:
          R = size(ALU_state.RAM[A_bus], set_CC);
          S = size(ALU_state.RAM[B_bus], set_CC);
          break;
     case ZQ:
          R = 0;
          S = size(ALU_state.Q, set_CC);
          break;
     case ZB:
          R = 0;
          S = size(ALU_state.RAM[B_bus], set_CC);
          break;
     case ZA:
          R = 0;
          S = size(ALU_state.RAM[A_bus], set_CC);
          break;
     case DA:
          R = size(bus.UIDB, set_CC);
          S = size(ALU_state.RAM[A_bus], set_CC);
          break;
     case DQ:
          R = size(bus.UIDB, set_CC);
          S = size(ALU_state.Q, set_CC);
          break;
     case DZ:
          R = size(bus.UIDB, set_CC);
          S = 0;
          break;
};

/* Perform the function corresponding to the function bits, and */
/* set the appropriate temporary flags */
switch(ALU_PL.funct) {
     case ADD:
     case SUBR:
     case SUBS:

          if (ALU_PL.funct == SUBR)
               R = ~R;
          if (ALU_PL.funct == SUBS)
               S = ~S;

          *F = R + S + Cin;

          *F = size(*F, set_CC);

          if (*F == 0)
               Z = 1;
          else
               Z = 0;

          N = (*F >> 31) & 0x00000001;
```

```
        if (((R >> 31) == (S >> 31)) && ((R >> 31) !=
                                        ((*F >> 31) & 0x00000001)))
             V = 1;
        else
             V = 0;

        if ((set_CC == 1) && (state.PL.data_ALU_mask == 1))
             switch (state.op_size) {
                  case 0:
                       C = (*F >> 8) & 0x01;
                       break;
                  case 1:
                       C = (*F >> 16) & 0x01;
                       break;
                  case 2:
                       C = ((((R & 0x0000ffff)
                            + (S & 0x0000ffff)) >> 16)
                            + (R >> 16) + (S >> 16)) >> 16;
                       break;
             }
        else
             C = ((((R & 0x0000ffff) + (S & 0x0000ffff)) >> 16)
                  + (R >> 16) + (S >> 16)) >> 16;

        X = C;
        break;
   case OR:
        *F = R | S;

        if (*F == 0)
             Z = 1;
        else
             Z = 0;

        N = (*F >> 31) & 0x00000001;

        V = 0;
        C = 0;
        X = 0;
        break;
   case AND:
   case NOTRS:
        if (ALU_PL.funct == NOTRS)
             R = ~R;

        *F = R & S;

        if (*F == 0)
             Z = 1;
        else
             Z = 0;

        N = (*F >> 31) & 0x00000001;

        V = 0;
        C = 0;
        X = 0;
        break;
   case EXOR:
```

```
        case EXNOR:

            *F = R ^ S;
            if (ALU_PL.funct == EXNOR)
                *F = ~*F;

            if (*F == 0)
                Z = 1;
            else
                Z = 0;

            N = (*F >> 31) & 0x00000001;

            V = 0;
            C = 0;
            X = 0;
            break;
    };

    /* Set the LIDB according to the OE */
    if (ALU_PL.OE == 1) {
        if (ALU_PL.dest == RAMA)
            bus.LIDB = ALU_state.RAM[A_bus];
        else
            bus.LIDB = *F;
    };

    if (set_CC == 1) {
        data_CC.C = C;
        data_CC.V = V;
        data_CC.Z = Z;
        data_CC.N = N;
        data_CC.X = X;
    }
    else {
        addr_CC.C = C;
        addr_CC.V = V;
        addr_CC.Z = Z;
        addr_CC.N = N;
        addr_CC.X = X;
    };

}
```

```
/*****************************************************************************/
/* size:                                                                     */
/*      Enables the data ALU slices as determined by the op_size             */
/* register.                                                                 */
/*****************************************************************************/

size(F, ALU_type)
int F;                          /* data */
int ALU_type;                   /* ALU type.  1=data  */
{
     int result;                /* temporary result */

     if ((state.PL.data_ALU_mask == 1) && (ALU_type == 1))
          switch (state.op_size) {
               case 0:
                    result = F & 0x000000ff;
                    if (((result >> 7) & 0x01) == 1)
                         result = result | 0xffffff00;
                    return (result);
                    break;
               case 1:
                    result = F & 0x0000ffff;
                    if (((result >> 15) & 0x01) == 1)
                         result = result | 0xffff0000;
                    return (result);
                    break;
               case 2:
                    return (F);
                    break;
          }
     return(F);
}
```

```
/*****************************************************************************/
/* dest_size:                                                                */
/*      Does the masking as specified by the size in the op_size             */
/* register.                                                                 */
/*****************************************************************************/

dest_size(reg,val)
int *reg;                   /* pointer to register */
int *val;                   /* pointer to value */
{
     if (state.PL.data_ALU_mask == 0)
          *reg = *val;
     else {
          switch (state.op_size) {
               case 0:
                    *reg = (*reg & 0xffffff00)
                              | (*val & 0x000000ff);
                    break;
               case 1:
                    *reg = (*reg & 0xffff0000)
                              | (*val & 0x0000ffff);
                    break;
               case 2:
                    *reg = *val;
                    break;
          }
     }
}
```

```
/*************************************************************************/
/* set_MM_busses:                                                        */
/*      Sets values onto the main memory control, address, and data      */
/* buses                                                                 */
/*************************************************************************/

set_MM_busses()
{
     /* Set the size lines according to the PL */
     switch(state.PL.MM.size_sel) {
          case 0:
               bus.MM_cntl.size = state.PL.MM.size;
               break;
          case 1:
               bus.MM_cntl.size = state.op_size;
               break;
          case 2:
               bus.MM_cntl.size = state.EWR & 0x00000003;
               break;
          case 3:
               bus.MM_cntl.size = (state.EWR >> 4) & 0x00000003;
               break;
     }

     /* Set the MM select and read lines according to the PL */
     bus.MM_cntl.select = state.PL.MM.select;
     bus.MM_cntl.read = state.PL.MM.read;

     /* Set the MM address bus to the MAR if its output is enabled */
     if (state.PL.MM.select == 1)
          bus.MM_address = state.MAR;

     /* If the MM is selected to be read, then set data bus */
     if ((bus.MM_cntl.select == 1) && (bus.MM_cntl.read == 1))
          switch (bus.MM_cntl.size) {
               case 0:
                    bus.MM_data = MM[bus.MM_address];
                    bus.MM_data = (bus.MM_data << 24) >> 24;
                    break;
               case 1:
                    bus.MM_data = (MM[bus.MM_address] << 24) >> 16;
                    bus.MM_data += MM[bus.MM_address + 1];
                    break;
               case 2:
               case 3:
                    bus.MM_data = MM[bus.MM_address] << 24;
                    bus.MM_data += (MM[bus.MM_address + 1] << 16);
                    bus.MM_data += (MM[bus.MM_address + 2] << 8);
                    bus.MM_data += MM[bus.MM_address + 3];
                    break;
          }

     /* Else, if the DBI is enabled, write its value onto the data bus */
     else if ((state.PL.MM.select & ~state.PL.MM.read) == 1)
          bus.MM_data = state.DBI;

     /* Else, clear the data bus */
     else
          bus.MM_data = 0;
}
```

```
/*****************************************************************************/
/* set_useq_bus:                                                             */
/*      Sets the microsequencer bus according to the PL select lines         */
/*****************************************************************************/

set_useq_bus()
{
     switch (state.PL.useq.map_sel) {
          case 0:
               bus.useq = state.PL.useq.NA;
               break;
          case 1:
               bus.useq = EA_PROM();
               break;
          case 2:
               state.IR = state.PIR;
               bus.useq = IR_PROM();
               break;
     }
}


/*****************************************************************************/
/* Include PROM:                                                             */
/*****************************************************************************/

#include "prom.c"

/*****************************************************************************/
/* main_memory:                                                              */
/*      Simulate any changes to the main memory                              */
/*****************************************************************************/

main_memory()
{
     /* If the MM is selected for a write operation, update it */
     if ((bus.MM_cntl.select == 1) && (bus.MM_cntl.read == 0))
          switch (bus.MM_cntl.size) {
               case 0:
                    MM[bus.MM_address] = bus.MM_data & 0x000000ff;
                    break;
               case 1:
                    MM[bus.MM_address] =
                      (bus.MM_data >> 8) & 0x000000ff;
                    MM[bus.MM_address + 1] =
                       bus.MM_data & 0x000000ff;
                    break;
               case 2:
               case 3:
                    MM[bus.MM_address] =
                      (bus.MM_data >> 24) & 0x000000ff;
                    MM[bus.MM_address + 1] =
                      (bus.MM_data >> 16) & 0x000000ff;
                    MM[bus.MM_address + 2] =
                      (bus.MM_data >> 8) & 0x000000ff;
                    MM[bus.MM_address + 3] =
                       bus.MM_data & 0x000000ff;
                    break;
          }
}
```

```
/*****************************************************************/
/* op_size:                                                      */
/*      Sets the op_size register                                */
/*****************************************************************/

op_size()
{
     if (state.PL.op_size.load == 1) {
          switch (state.PL.op_size.select) {
               case 0:
                    state.op_size = state.PL.op_size.data;
                    break;
               case 1:
                    state.op_size = (state.IR >> 6) & 0x00000003;
                    break;
               case 2:
                    state.op_size = (state.IR >> 9) & 0x00000003;
                    break;
               case 3:
                    state.op_size = (state.IR >> 12) & 0x00000003;
                    break;
          };
     };
}


/*****************************************************************/
/* useq:                                                         */
/*      Sets the uPC according to the microsequencer instruction */
/*****************************************************************/

useq()
{
     int top;                  /* stack top index */

     switch (state.PL.useq.inst) {
          case CJS:
               /* First check the CC, and then set the uPC */
               if (state.useq.CC == TRUE) {
                    push(state.useq.uPC + 1);
                    state.useq.uPC = bus.useq;
               }
               else
                    state.useq.uPC++;
               break;
          case JMAP:
               /* Find starting address of routine to implement */
               /* the machine instruction in the IR */
               state.useq.uPC = bus.useq;
               break;
          case CJP:
               /* First check the CC, and then set the uPC */
               if (state.useq.CC == TRUE)
                    state.useq.uPC = bus.useq;
               else
                    state.useq.uPC ++;
               break;
```

```
        case PUSH:
            /* First check the CC, and then set stack and counter */
            push(state.useq.uPC + 1);
            if (state.useq.CC == TRUE)
                state.useq.counter = bus.useq;
            state.useq.uPC ++;
            break;
        case CRTN:
            /* First check the CC, and then set the uPC */
            if (state.useq.CC == TRUE) {
                top = state.useq.stack.top;
                if (top == 0)
                    state.useq.uPC = state.useq.stack.element[0];
                else {
                    state.useq.uPC = state.useq.stack.element[top-1];
                    --state.useq.stack.top;
                }
            }
            else
                state.useq.uPC ++;
            break;
        case CONT:
            state.useq.uPC ++;
            break;
        case TWB:
            if (state.useq.CC == TRUE) {
                state.useq.uPC ++;
                if (state.useq.stack.top != 0)
                    --state.useq.stack.top;
            }
            else {
                if (state.useq.counter != 0) {
                    --state.useq.counter;
                    top = state.useq.stack.top;
                    state.useq.uPC = state.useq.stack.element[top - 1];
                }
                else {
                    state.useq.uPC = bus.useq;
                    if (state.useq.stack.top != 0)
                        --state.useq.stack.top;
                }
            }
            break;
    };
}
```

```
/*****************************************************************************/
/* useq_CC:                                                                  */
/*      Determines whether the CC input to the microsequencer is TRUE        */
/* or FALSE                                                                  */
/*****************************************************************************/

useq_CC()
{
     int useq_CC[4];                /* CC array */
     int shift;                     /* amount to shift */
     int index;                     /* index into useq_CC array */
     int CC_output;                 /* output from CC mux */

     int C;                         /* temporary condition codes */
     int Z;
     int V;
     int N;
     int X;

     int Bcc;                      /* Bcc circuit output */

     /* Set the temporary condition codes */
     C = state.TCC.C;
     Z = state.TCC.Z;
     V = state.TCC.V;
     N = state.TCC.N;
     X = state.TCC.X;

     /* Calculate the Bcc circuit output */
     switch ((state.IR >> 8) & 0x0f) {
          case 2:
               Bcc = (~C & ~Z) & 0x00000001;
               break;
          case 3:
               Bcc = C | Z;
               break;
          case 4:
               Bcc = ~C & 0x00000001;
               break;
          case 5:
               Bcc = C;
               break;
          case 6:
               Bcc = ~Z & 0x00000001;
               break;
          case 7:
               Bcc = Z;
               break;
          case 8:
               Bcc = ~V & 0x00000001;
               break;
          case 9:
               Bcc = V;
               break;
          case 10:
               Bcc = ~N & 0x00000001;
               break;
```

```
            case 11:
                Bcc = N;
                break;
            case 12:
                Bcc = ((N & V) | (~N & ~V)) & 0x00000001;
                break;
            case 13:
                Bcc = N ^ V;
                break;
            case 14:
                Bcc = ((N & V & ~Z) | (~N & ~V & ~Z)) & 0x00000001;
                break;
            case 15:
                Bcc = (N ^ V) | Z;
                break;
        }

        /* Set up the useq_CC array. */
        useq_CC[0] = state.EWR + (state.IR << 16);
        useq_CC[1] = state.LIDB;
        useq_CC[2] = Bcc + (Bcc << 1) + (Bcc << 2)
                     + (Bcc << 3) + (Bcc << 4) + (Bcc << 5)
                     + (Bcc << 6) + (Bcc << 7) + (Bcc << 8) + (Bcc << 9)
                     + (Bcc << 10) + (Bcc << 11) + (Bcc << 12)
                     + (Bcc << 13) + (Bcc << 14) + (Bcc << 15)
                     + (state.TCC.Z << 16) + (TRUE << 19) + (FALSE << 20)
                     + (state.HCC.Z << 21) + (state.HCC.N << 22)
                     + (state.HCC.V << 23) + (state.HCC.C << 24);
        useq_CC[3] = 0;

        /* find the condition code bit to be returned */
        index = state.PL.useq.CC_sel/32;
        CC_output = useq_CC[index] >> (state.PL.useq.CC_sel & 0x0000001f);
        CC_output = (CC_output & 0x00000001);

        /* check to see which logic level is true and return result */
        if (CC_output == state.PL.useq.CC_logic_level)
            return(TRUE);
        else
            return(FALSE);
}
```

```
/*************************************************************************/
/* push:                                                                 */
/*      Pushes an element onto the microsequencer stack                  */
/*************************************************************************/

push(what_to_push)
int what_to_push;                   /* what to push */
{
     int top;                       /* top of stack index */

     top = state.useq.stack.top;
     if (top < 9) {
          state.useq.stack.element[top] =  what_to_push;
          state.useq.stack.top++;
     }
     else
          state.useq.stack.element[top-1] = what_to_push;
}


/*************************************************************************/
/* print_host_state:                                                     */
/*      Prints the current state of the host.                            */
/*************************************************************************/

print_host_state()
{
     int i;                         /* index */
     int ch;                        /* temporary character */
     char hex_string1[100];         /* temporary string */


     printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
     print_PL(state.PL);

     printf("\n");
     printf("Press <RETURN> to continue. ");
     while (getchar() != '\n')
          ;

     printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
     printf("Buses:\n");
     hex(bus.UIDB, 8);
     printf("UIDB            = %s\n", hex_string);
     hex(bus.LIDB, 8);
     printf("LIDB            = %s\n", hex_string);
     hex(bus.MM_data, 8);
     printf("MM_data         = %s\n", hex_string);
     hex(bus.MM_address, 8);
     printf("MM_address      = %s\n", hex_string);
     printf("MM_cntl.size    =        %x\n",bus.MM_cntl.size);
     printf("MM_cntl.select  =        %x\n",bus.MM_cntl.select);
     printf("MM_cntl.read    =        %x\n",bus.MM_cntl.read);
     printf("data_ALU_A      =        %x\n",bus.data_ALU_A);
     printf("data_ALU_B      =        %x\n",bus.data_ALU_B);
     printf("addr_ALU_A      =        %x\n",bus.addr_ALU_A);
     printf("addr_ALU_B      =        %x\n",bus.addr_ALU_B);
     hex(bus.useq, 3);
     printf("useq            =      %s\n", hex_string);
```

```
printf("\n");
printf("Press <RETURN> to continue. ");
while (getchar() != '\n')
    ;
printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");

printf("Registers:\n\n");
hex(state.DBI, 8);
printf("   DBI              =  %s\n", hex_string);
hex(state.MAR, 8);
printf("   MAR              =  %s\n", hex_string);
hex(state.PIR, 4);
printf("   PIR              =      %s\n", hex_string);
hex(state.IR, 4);
printf("   IR               =      %s\n", hex_string);
hex(state.EWR, 4);
printf("   EWR              =      %s\n", hex_string);
hex(state.LIDB, 8);
printf("   LIDB             =  %s\n", hex_string);
printf("   op_size          =         %x\n", state.op_size);
hex(state.mult.data_1, 8);
printf("   mult.data_1      =  %s\n", hex_string);
hex(state.mult.data_2, 8);
printf("   mult.data_2      =  %s\n", hex_string);
hex(state.shifter.shift, 2);
printf("   shifter.shift    =        %s\n", hex_string);
hex(state.shifter.data, 8);
printf("   shifter.data     =  %s\n", hex_string);
printf("   shifter.X        =         %x\n", state.shifter.X);
printf("   shifter.N        =         %x\n", state.shifter.N);
printf("   shifter.Z        =         %x\n", state.shifter.Z);
printf("   shifter.C        =         %x\n", state.shifter.C);
hex(state.feedback, 8);
printf("   feedback         =  %s\n", hex_string);
hex(state.EAR, 2);
printf("   EAR              =        %s\n", hex_string);

printf("Press <RETURN> to continue. ");
while (getchar() != '\n')
    ;
printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
```

```
printf("Microsequencer:\n");
hex(state.useq.uPC, 3);
printf("   uPC      =   %s\n", hex_string);
hex(state.useq.counter, 3);
printf("   Counter =   %s\n", hex_string);
hex(state.PL.useq.CC_sel, 2);
printf("   CC_sel  =    %s\n", hex_string);
if (state.useq.CC == TRUE)
     printf("   CC       =  TRUE\n");
else
     printf("   CC       = FALSE\n");
printf("   Stack:\n");
for(i=0;i<state.useq.stack.top;i++) {
     hex(state.useq.stack.element[i],3);
     if (i == state.useq.stack.top-1) {
          printf("      TOP --> %s\n", hex_string);
     }
     else
          printf("              %s\n", hex_string);
};

printf("\n");
printf("Target Flags:\n");
printf("   C = %x   Z = %x   V = %x   N = %x   X = %x\n\n",
       state.TCC.C,state.TCC.Z,state.TCC.V,state.TCC.N,state.TCC.X);

printf("\n");
printf("Host Flags:\n");
printf("   C = %x   Z = %x   V = %x   N = %x   X = %x\n\n",
       state.HCC.C,state.HCC.Z,state.HCC.V,state.HCC.N,state.HCC.X);

printf("Press <RETURN> to continue. ");
while (getchar() != '\n')
     ;
printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");

printf("Data ALU:                                Address ALU:\n");
for(i=0;i<10;i++) {
     hex(state.data_ALU.RAM[i], 8);
     strcpy(hex_string1, hex_string);
     hex(state.addr_ALU.RAM[i], 8);
     printf("   RAM %d   = %s                    RAM %d   = %s\n",
                                  i,hex_string1,i,hex_string);
}
for(i=10;i<16;i++) {
     hex(state.data_ALU.RAM[i], 8);
     strcpy(hex_string1, hex_string);
     hex(state.addr_ALU.RAM[i], 8);
     printf("   RAM %d  = %s                    RAM %d  = %s\n",
                                  i,hex_string1,i,hex_string);
}
hex(state.data_ALU.Q, 8);
strcpy(hex_string1, hex_string);
hex(state.addr_ALU.Q, 8);
printf("   Q        = %s                    Q        = %s\n",
                                  hex_string1,hex_string);
```

```
      printf("   Source       = %x                    Source       = %x\n",
            state.PL.data_ALU.src,state.PL.addr_ALU.src);
      printf("   Function     = %x                    Function     = %x\n",
            state.PL.data_ALU.funct,state.PL.addr_ALU.funct);
      printf("   Destination  = %x                    Destination  = %x\n",
            state.PL.data_ALU.dest,state.PL.addr_ALU.dest);
      printf("   OE           = %x                    OE           = %x\n\n",
            state.PL.data_ALU.OE,state.PL.addr_ALU.OE);

      printf("Press <RETURN> to continue. ");
      while (getchar() != '\n')
            ;
      printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
}


/*************************************************************************/
/* print_PL:                                                             */
/*      Print the PL                                                     */
/*************************************************************************/

print_PL(uinst)
struct PL_type uinst;          /* microinstruction to be printed */
{
      hex(uinst.useq.NA, 3);
      printf("useq.NA ........... %s    ", hex_string);
      printf("addr_ALU inst ... %x %x %x    ", uinst.addr_ALU.dest,
            uinst.addr_ALU.funct, uinst.addr_ALU.src);
      printf("Cin.select ......... %x\n", uinst.Cin.select);

      printf("useq.inst ......... %x    ", uinst.useq.inst);
      printf("addr_ALU.OE ....... %x    ", uinst.addr_ALU.OE);
      printf("mult.data_1_load .... %x\n", uinst.mult.data_1_load);

      printf("useq.map_sel ...... %x    ", uinst.useq.map_sel);
      printf("addr_ALU.A_bus .... %x    ", uinst.addr_ALU.A_bus);
      printf("mult.data_2_load .... %x\n", uinst.mult.data_2_load);

      printf("useq.CC_logic_level . %x    ", uinst.useq.CC_logic_level);
      printf("addr_ALU.A_bus_sel .. %x    ", uinst.addr_ALU.A_bus_sel);
      printf("shifter.inst ....... %x\n", uinst.shifter.inst);

      hex(uinst.useq.CC_sel, 2);
      printf("useq.CC_sel ....... %s    ", hex_string);
      printf("addr_ALU.B_bus .... %x    ", uinst.addr_ALU.B_bus);
      printf("shifter.shift_load .. %x\n", uinst.shifter.shift_load);

      printf("IR_load ........... %x    ", uinst.IR_load);
      printf("addr_ALU.B_bus_sel .. %x    ", uinst.addr_ALU.B_bus_sel);
      printf("shifter.data_load ... %x\n", uinst.shifter.data_load);

      printf("EWR_load ........... %x    ", uinst.EWR_load);
      printf("                          ");
      printf("shifter.CX_load ..... %x\n", uinst.shifter.CX_load);

      printf("                          ");
      printf("MAR.load ........... %x    ", uinst.MAR.load);
      printf("feedback.load ..... %x\n", uinst.feedback.load);
```

```
    printf("                                ");
    printf("const_addr ..... %x     ", uinst.const_addr);
    printf("feedback.size_sel ... %x\n", uinst.feedback.size_sel);

    printf("op_size.load ..... %x     ", uinst.op_size.load);
    printf("                               ");
    printf("feedback.size ..... %x\n", uinst.feedback.size);

    printf("op_size.select ..... %x     ", uinst.op_size.select);
    printf("DBI.load ..... %x     ", uinst.DBI.load);
    printf("feedback.scale_sel .. %x\n", uinst.feedback.scale_sel);

    printf("op_size.data ..... %x     ", uinst.op_size.data);
    printf("                              ");
    printf("EAR.load ..... %x\n", uinst.EAR.load);

    printf("UIDB_sel ..... %x     ", uinst.UIDB_sel);
    printf("MM.select ..... %x     ", uinst.MM.select);
    printf("EAR.select ..... %x\n", uinst.EAR.select);

    printf("data_ALU inst ... %x %x %x     ", uinst.data_ALU.dest,
           uinst.data_ALU.funct, uinst.data_ALU.src);
    printf("MM.read ..... %x     ", uinst.MM.read);
    printf("data_ALU_mask ..... %x\n", uinst.data_ALU_mask);

    printf("data_ALU.OE ..... %x     ", uinst.data_ALU.OE);
    printf("MM.size_sel ..... %x\n", uinst.MM.size_sel);

    printf("data_ALU.A_bus ..... %x     ", uinst.data_ALU.A_bus);
    printf("MM.size ..... %x\n", uinst.MM.size);

    printf("data_ALU.A_bus_sel .. %x     ", uinst.data_ALU.A_bus_sel);
    printf("TCC.XNZVC_load .. %x%x%x%x%x\n", uinst.TCC.X_load,
           uinst.TCC.N_load, uinst.TCC.Z_load, uinst.TCC.V_load,
           uinst.TCC.C_load);

    printf("data_ALU.B_bus ..... %x     ", uinst.data_ALU.B_bus);
    printf("SR_sel ..... %x\n", uinst.SR_sel);

    printf("data_ALU.B_bus_sel .. %x     ", uinst.data_ALU.B_bus_sel);
    printf("Cin.data ..... %x\n", uinst.Cin.data);
}
```

```
/*************************************************************************/
/* print_target_state:                                                   */
/*      Prints the current state of the target machine                   */
/*************************************************************************/

print_target_state()
{
     int i;                         /* index */
     int ch;                        /* temporary character */
     char hex_string1[100];         /* temporary string  */

     printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
     printf("Buses:\n");
     hex(bus.MM_address, 8);
     printf("   MM_select  = %x                    MM_address   = %s\n",
            bus.MM_cntl.select, hex_string);
     hex(bus.MM_data, 8);
     printf("   MM_read    = %x                    MM_data      = %s\n",
            bus.MM_cntl.read, hex_string);
     printf("   MM_size    = %x\n", bus.MM_cntl.size);

     printf("\n");
     printf("Registers:\n");
     hex(state.addr_ALU.RAM[8], 8);
     strcpy(hex_string1, hex_string);
     hex(state.MAR, 8);
     printf("   PC       = %s                MAR          = %s\n",
            hex_string1, hex_string);

     hex(state.addr_ALU.RAM[9], 8);
     strcpy(hex_string1, hex_string);
     hex(state.addr_ALU.RAM[10], 8);
     printf("   A7'      = %s                A7''         = %s\n",
           hex_string1, hex_string);

     hex(state.IR, 8);
     strcpy(hex_string1, hex_string);
     hex(state.EWR, 8);
     printf("   IR       = %s                EWR          = %s\n",
            hex_string1, hex_string);

     printf("\n");
     printf("Target Flags:\n");
     printf("   C = %x   Z = %x   V = %x   N = %x   X = %x\n",
           state.TCC.C,state.TCC.Z,state.TCC.V,state.TCC.N,state.TCC.X);

     printf("\n");
     printf("Data Registers:                      Address Registers:\n");
     for(i=0;i<8;i++) {
          hex(state.data_ALU.RAM[i], 8);
          strcpy(hex_string1, hex_string);
          hex(state.addr_ALU.RAM[i], 8);
          printf("   D%d   = %s                        A%d   = %s\n",
                                    i,hex_string1,i,hex_string);
     }
     printf("\n");
     printf("Press <RETURN> to continue. ");
     while (getchar() != '\n');
     printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
}
```

```
/*****************************************************************************/
/* shifter:                                                                  */
/*      Returns the result of the shifter                                    */
/*****************************************************************************/

shifter()
{
     int shift;                    /* shift amount */
     int i;                        /* counter */
     unsigned int result;          /* shifter result */
     int sign_bit;                 /* sign bit */

     shift = state.shifter.shift;
     result = state.shifter.data;
     state.shifter.V = 0;
     switch(state.op_size) {
          case 0:
               switch(state.PL.shifter.inst) {
                    case 0:
                         for (i=0; i<shift; i++) {
                              state.shifter.C = result & 0x01;
                              state.shifter.X = state.shifter.C;
                              result = (result & 0xffffff00) |
                                       ((result >> 1) & 0x0000007f) |
                                       (result & 0x00000080) ;
                         }
                         break;
                    case 1:
                         sign_bit = result & 0x080;
                         for (i=0; i<shift; i++) {
                              state.shifter.C = (result >> 7) & 0x01;
                              state.shifter.X = state.shifter.C;
                              result = (result & 0xffffff00) |
                                       ((result << 1) & 0x000000ff);
                              if ((result & 0x080) != sign_bit)
                                   state.shifter.V = 1;
                         }
                         break;
                    case 2:
                         for (i=0; i<shift; i++) {
                              state.shifter.C = result & 0x01;
                              state.shifter.X = state.shifter.C;
                              result = (result & 0xffffff00) |
                                       ((result >> 1) & 0x0000007f);
                         }
                         break;
```

```
            case 3:
                for (i=0; i<shift; i++) {
                    state.shifter.C = (result >> 7) & 0x01;
                    state.shifter.X = state.shifter.C;
                    result = (result & 0xffffff00) |
                             ((result << 1) & 0x000000ff);
                }
                break;
            case 4:
                for (i=0; i<shift; i++) {
                    state.shifter.C = result & 0x01;
                    result = (result & 0xffffff00) |
                             ((result >> 1) & 0x07f) |
                             (state.shifter.X * 0x080);
                    state.shifter.X = state.shifter.C;
                }
                break;
            case 5:
                for (i=0; i<shift; i++) {
                    state.shifter.C = (result >> 7) & 0x01;
                    result = (result & 0xffffff00) |
                             ((result << 1) & 0x0ff) |
                             (state.shifter.X);
                    state.shifter.X = state.shifter.C;
                }
                break;
            case 6:
                for (i=0; i<shift; i++) {
                    state.shifter.C = result & 0x01;
                    result = (result & 0xffffff00) |
                             ((result >> 1) & 0x07f) |
                             (state.shifter.C * 0x080);
                }
                break;
            case 7:
                for (i=0; i<shift; i++) {
                    state.shifter.C = (result >> 7) & 0x01;
                    result = (result & 0xffffff00) |
                             ((result << 1) & 0x0ff) |
                             (state.shifter.C);
                }
                break;
        };
        if (result == 0)
            state.shifter.Z = 1;
        else
            state.shifter.Z = 0;
        state.shifter.N = result >> 31;
        break;
    case 1:
        switch(state.PL.shifter.inst) {
            case 0:
                for (i=0; i<shift; i++) {
                    state.shifter.C = result & 0x01;
                    state.shifter.X = state.shifter.C;
                    result = (result & 0xffff0000) |
                             ((result >> 1) & 0x00007fff) |
                             (result & 0x00008000) ;
                }
                break;
```

```
case 1:
     for (i=0; i<shift; i++) {
          sign_bit = result & 0x08000;
          state.shifter.C = (result >> 15) & 0x01;
          state.shifter.X = state.shifter.C;
          result = (result & 0xffff0000) |
                   ((result << 1) & 0x0000ffff);
          if ((result & 0x08000) != sign_bit)
               state.shifter.V = 1;
     }
     break;
case 2:
     for (i=0; i<shift; i++) {
          state.shifter.C = result & 0x01;
          state.shifter.X = state.shifter.C;
          result = (result & 0xffff0000) |
                   ((result >> 1) & 0x00007fff);
     }
     break;
case 3:
     for (i=0; i<shift; i++) {
          state.shifter.C = (result >> 15) & 0x01;
          state.shifter.X = state.shifter.C;
          result = (result & 0xffff0000) |
                   ((result << 1) & 0x0000ffff);
     }
     break;
case 4:
     for (i=0; i<shift; i++) {
          state.shifter.C = result & 0x01;
          result = (result & 0xffff0000) |
                   ((result >> 1) & 0x07fff) |
                   (state.shifter.X * 0x08000);
          state.shifter.X = state.shifter.C;
     }
     break;
case 5:
     for (i=0; i<shift; i++) {
          state.shifter.C = (result >> 15) & 0x01;
          result = (result & 0xffff0000) |
                   ((result << 1) & 0x0ffff) |
                   (state.shifter.X);
          state.shifter.X = state.shifter.C;
     }
     break;
case 6:
     for (i=0; i<shift; i++) {
          state.shifter.C = result & 0x01;
          result = (result & 0xffff0000) |
                   ((result >> 1) & 0x07fff) |
                   (state.shifter.C * 0x08000);
     }
     break;
```

```
                case 7:
                    for (i=0; i<shift; i++) {
                        state.shifter.C = (result >> 15) & 0x01;
                        result = (result & 0xffff0000) |
                                 ((result << 1) & 0x0ffff) |
                                 (state.shifter.C);
                    }
                    break;
            };
            if (result == 0)
                state.shifter.Z = 1;
            else
                state.shifter.Z = 0;
            state.shifter.N = result >> 31;
            break;
        case 2:
            switch(state.PL.shifter.inst) {
                case 0:
                    for (i=0; i<shift; i++) {
                        state.shifter.C = result & 0x01;
                        state.shifter.X = state.shifter.C;
                        result = (result >> 1) |
                                 (result & 0x80000000);
                    }
                    break;
                case 1:
                    for (i=0; i<shift; i++) {
                        sign_bit = result & 0x80000000;
                        state.shifter.C = (result >> 31) & 0x01;
                        state.shifter.X = state.shifter.C;
                        result = result << 1;
                        if ((result & 0x080000000) != sign_bit)
                            state.shifter.V = 1;
                    }
                    break;
                case 2:
                    for (i=0; i<shift; i++) {
                        state.shifter.C = result & 0x01;
                        state.shifter.X = state.shifter.C;
                        result = result >> 1;
                    }
                    break;
                case 3:
                    for (i=0; i<shift; i++) {
                        state.shifter.C = (result >> 31) & 0x01;
                        state.shifter.X = state.shifter.C;
                        result = result << 0x01;
                    }
                    break;
                case 4:
                    for (i=0; i<shift; i++) {
                        state.shifter.C = result & 0x01;
                        result = (result >> 1) |
                                 (state.shifter.X << 31);
                        state.shifter.X = state.shifter.C;
                    }
                    break;
```

```
                    case 5:
                        for (i=0; i<shift; i++) {
                            state.shifter.C = (result >> 31) & 0x01;
                            result = (result << 1) | state.shifter.X;
                            state.shifter.X = state.shifter.C;
                        }
                        break;
                    case 6:
                        for (i=0; i<shift; i++) {
                            state.shifter.C = result & 0x01;
                            result = (result >> 1) | (result << 31);
                        }
                        break;
                    case 7:
                        for (i=0; i<shift; i++) {
                            state.shifter.C = (result >> 31) & 0x01;
                            result = (result << 1) | state.shifter.C;
                        }
                        break;
                };
                if (result == 0)
                    state.shifter.Z = 1;
                else
                    state.shifter.Z = 0;
                state.shifter.N = result >> 31;
                break;
        };
        return(result);
}
```

```
/*****************************************************************************/
/* EA_PROM:                                                                  */
/*****************************************************************************/

EA_PROM()
{
     int mode;                          /* operand mode */
     int reg;                           /* operand register */
     int EA;                            /* effective address */

     int EWR_8;                         /* EWR bit 8 */
     int EWR_15;                        /* EWR bit 15 */
     int BS;                            /* Base Suppress */
     int BDS;                           /* Base Disp. Suppress */
     int ODS;                           /* Outer Disp. Suppress */
     int IS;                            /* Index Suppress */

     int POSTI;                         /* Post Index */
     int PREI;                          /* Pre Index */

     EWR_8 = (state.EWR >> 8) & 0x01;
     EWR_15 = (state.EWR >> 15) & 0x01;
     BS = (state.EWR >> 7) & 0x01;
     IS = (state.EWR >> 6) & 0x01;
     BDS = (((state.EWR >> 4) & 0x03) == 1);
     ODS = ((state.EWR & 0x03) == 1);

     if ((((state.EWR >> 2) & 0x01) == 1) && !IS)
          POSTI = 1;
     else
          POSTI = 0;
     if ((((state.EWR >> 2) & 0x01) == 0) && !IS)
          PREI = 1;
     else
          PREI = 0;

     if (state.PL.EAR.select == 0) {
          mode = (state.PIR >> 3) & 0x07;
          reg = state.PIR & 0x07;
     }
     else {
          mode = (state.PIR >> 6) & 0x07;
          reg = (state.PIR >> 9) & 0x07;
     }

     if (mode == 0)
          return (0x010);
     if (mode == 1)
          return (0x012);
     if (mode == 2)
          return (0x014);
     if (mode == 3)
          return (0x016);
     if (mode == 4)
          return (0x019);
     if (mode == 5)
          return (0x01c);
```

```
if (mode == 6) {
     if (!EWR_8 && !EWR_15)
          return (0x01f);
     if (!EWR_8 && EWR_15)
          return (0x023);
     if (EWR_8 && !POSTI && !PREI) {
          if (BDS && BS && IS)
               return (0x028);
          if (!BDS && BS && IS)
               return (0x02b);
          if (BDS && !BS && IS)
               return (0x014);
          if (!BDS && !BS && IS)
               return (0x037);
          if (BDS && BS && !IS && !EWR_15)
               return (0x030);
          if (BDS && BS && !IS && EWR_15)
               return (0x033);
          if (!BDS && BS && !IS && !EWR_15)
               return (0x03c);
          if (!BDS && BS && !IS && EWR_15)
               return (0x042);
          if (BDS && !BS && !IS && !EWR_15)
               return (0x049);
          if (BDS && !BS && !IS && EWR_15)
               return (0x04c);
          if (!BDS && !BS && !IS && !EWR_15)
               return (0x04f);
          if (!BDS && !BS && !IS && EWR_15)
               return (0x055);
     }
     if (EWR_8 && POSTI && !PREI) {
          if (BDS && BS && IS && ODS)
               return (0x05c);
          if (!BDS && BS && IS && ODS)
               return (0x060);
          if (BDS && !BS && IS && ODS)
               return (0x067);
          if (!BDS && !BS && IS && ODS)
               return (0x070);
          if (BDS && BS && !IS && !EWR_15 && ODS)
               return (0x030);
          if (BDS && BS && !IS && EWR_15 && ODS)
               return (0x033);
          if (!BDS && BS && !IS && !EWR_15 && ODS)
               return (0x077);
          if (!BDS && BS && !IS && EWR_15 && ODS)
               return (0x07e);
          if (BDS && !BS && !IS && !EWR_15 && ODS)
               return (0x090);
          if (BDS && !BS && !IS && EWR_15 && ODS)
               return (0x095);
          if (!BDS && !BS && !IS && !EWR_15 && ODS)
               return (0x0af);
          if (!BDS && !BS && !IS && EWR_15 && ODS)
               return (0x0b6);
          if (BDS && BS && IS && !ODS)
               return (0x06b);
```

```
        if (!BDS && BS && IS && !ODS)
             return (0x086);
        if (BDS && !BS && IS && !ODS)
             return (0x09b);
        if (!BDS && !BS && IS && !ODS)
             return (0x0be);
        if (BDS && BS && !IS && !EWR_15 && !ODS)
             return (0x0a2);
        if (BDS && BS && !IS && EWR_15 && !ODS)
             return (0x0a8);
        if (!BDS && BS && !IS && !EWR_15 && !ODS)
             return (0x220);
        if (!BDS && BS && !IS && EWR_15 && !ODS)
             return (0x225);
        if (BDS && !BS && !IS && !EWR_15 && !ODS)
             return (0x0c8);
        if (BDS && !BS && !IS && EWR_15 && !ODS)
             return (0x0d0);
        if (!BDS && !BS && !IS && !EWR_15 && !ODS)
             return (0x0d9);
        if (!BDS && !BS && !IS && EWR_15 && !ODS)
             return (0x0e4);
   }
   if (EWR_8 && !POSTI && PREI) {
        if (BDS && BS && IS && ODS)
             return (0x05c);
        if (!BDS && BS && IS && ODS)
             return (0x060);
        if (BDS && !BS && IS && ODS)
             return (0x067);
        if (!BDS && !BS && IS && ODS)
             return (0x070);
        if (BDS && BS && !IS && !EWR_15 && ODS)
             return (0x0f0);
        if (BDS && BS && !IS && EWR_15 && ODS)
             return (0x0f5);
        if (!BDS && BS && !IS && !EWR_15 && ODS)
             return (0x0fa);
        if (!BDS && BS && !IS && EWR_15 && ODS)
             return (0x102);
        if (BDS && !BS && !IS && !EWR_15 && ODS)
             return (0x10b);
        if (BDS && !BS && !IS && EWR_15 && ODS)
             return (0x110);
        if (!BDS && !BS && !IS && !EWR_15 && ODS)
             return (0x126);
        if (!BDS && !BS && !IS && EWR_15 && ODS)
             return (0x12e);
        if (BDS && BS && IS && !ODS)
             return (0x06b);
        if (!BDS && BS && IS && !ODS)
             return (0x086);
        if (BDS && !BS && IS && !ODS)
             return (0x09b);
        if (!BDS && !BS && IS && !ODS)
             return (0x0be);
        if (BDS && BS && !IS && !EWR_15 && !ODS)
             return (0x115);
```

```
            if (BDS && BS && !IS && EWR_15 && !ODS)
                return (0x11d);
            if (!BDS && BS && !IS && !EWR_15 && !ODS)
                return (0x22a);
            if (!BDS && BS && !IS && EWR_15 && !ODS)
                return (0x22f);
            if (BDS && !BS && !IS && !EWR_15 && !ODS)
                return (0x137);
            if (BDS && !BS &&   !IS && EWR_15 && !ODS)
                return (0x13f);
            if (!BDS && !BS && !IS && !EWR_15 && !ODS)
                return (0x148);
            if (!BDS && !BS && !IS && EWR_15 && !ODS)
                return (0x153);
        }
    }
    if ((mode == 7) && (reg == 0))
        return (0x215);
    if ((mode == 7) && (reg == 1))
        return (0x219);
    if ((mode == 7) && (reg == 2))
        return (0x15f);
    if ((mode == 7) && (reg == 3)) {
        if (!EWR_8 && !EWR_15)
            return (0x162);
        if (!EWR_8 && EWR_15)
            return (0x166);
        if (EWR_8 && !POSTI && !PREI) {
            if (BDS && BS && IS)
                return (0x028);
            if (!BDS && BS && IS)
                return (0x02b);
            if (BDS && !BS && IS)
                return (0x16b);
            if (!BDS && !BS && IS)
                return (0x16d);
            if (BDS && BS && !IS && !EWR_15)
                return (0x030);
            if (BDS && BS && !IS && EWR_15)
                return (0x033);
            if (!BDS && BS && !IS && !EWR_15)
                return (0x03c);
            if (!BDS && BS && !IS && EWR_15)
                return (0x042);
            if (BDS && !BS && !IS && !EWR_15)
                return (0x172);
            if (BDS && !BS && !IS && EWR_15)
                return (0x175);
            if (!BDS && !BS && !IS && !EWR_15)
                return (0x178);
            if (!BDS && !BS && !IS && EWR_15)
                return (0x17e);
        }
```

```
if (EWR_8 && POSTI && !PREI) {
     if (BDS && BS && IS && ODS)
          return (0x05c);
     if (!BDS && BS && IS && ODS)
          return (0x060);
     if (BDS && !BS && IS && ODS)
          return (0x185);
     if (!BDS && !BS && IS && ODS)
          return (0x189);
     if (BDS && BS && !IS && !EWR_15 && ODS)
          return (0x030);
     if (BDS && BS && !IS && EWR_15 && ODS)
          return (0x033);
     if (!BDS && BS && !IS && !EWR_15 && ODS)
          return (0x077);
     if (!BDS && BS && !IS && EWR_15 && ODS)
          return (0x07e);
     if (BDS && !BS && !IS && !EWR_15 && ODS)
          return (0x190);
     if (BDS && !BS && !IS && EWR_15 && ODS)
          return (0x195);
     if (!BDS && !BS && !IS && !EWR_15 && ODS)
          return (0x1a2);
     if (!BDS && !BS && !IS && EWR_15 && ODS)
          return (0x1a9);
     if (BDS && BS && IS && !ODS)
          return (0x06b);
     if (!BDS && BS && IS && !ODS)
          return (0x086);
     if (BDS && !BS && IS && !ODS)
          return (0x19b);
     if (!BDS && !BS && IS && !ODS)
          return (0x1b1);
     if (BDS && BS && !IS && !EWR_15 && !ODS)
          return (0x0a2);
     if (BDS && BS && !IS && EWR_15 && !ODS)
          return (0x0a8);
     if (!BDS && BS && !IS && !EWR_15 && !ODS)
          return (0x220);
     if (!BDS && BS && !IS && EWR_15 && !ODS)
          return (0x225);
     if (BDS && !BS && !IS && !EWR_15 && !ODS)
          return (0x1bb);
     if (BDS && !BS && !IS && EWR_15 && !ODS)
          return (0x1c3);
     if (!BDS && !BS && !IS && !EWR_15 && !ODS)
          return (0x1cc);
     if (!BDS && !BS && !IS && EWR_15 && !ODS)
          return (0x1d7);
}
```

```
        if (EWR_8 && !POSTI && PREI) {
            if (BDS && BS && IS && ODS)
                return (0x05c);
            if (!BDS && BS && IS && ODS)
                return (0x060);
            if (BDS && !BS && IS && ODS)
                return (0x185);
            if (!BDS && !BS && IS && ODS)
                return (0x189);
            if (BDS && BS && !IS && !EWR_15 && ODS)
                return (0x0f0);
            if (BDS && BS && !IS && EWR_15 && ODS)
                return (0x0f5);
            if (!BDS && BS && !IS && !EWR_15 && ODS)
                return (0x0fa);
            if (!BDS && BS && !IS && EWR_15 && ODS)
                return (0x102);
            if (BDS && !BS && !IS && !EWR_15 && ODS)
                return (0x1e3);
            if (BDS && !BS && !IS && EWR_15 && ODS)
                return (0x1e8);
            if (!BDS && !BS && !IS && !EWR_15 && ODS)
                return (0x1ed);
            if (!BDS && !BS && !IS && EWR_15 && ODS)
                return (0x1f5);
            if (BDS && BS && IS && !ODS)
                return (0x06b);
            if (!BDS && BS && IS && !ODS)
                return (0x086);
            if (BDS && !BS && IS && !ODS)
                return (0x19b);
            if (!BDS && !BS && IS && !ODS)
                return (0x1b1);
            if (BDS && BS && !IS && !EWR_15 && !ODS)
                return (0x115);
            if (BDS && BS && !IS && EWR_15 && !ODS)
                return (0x11d);
            if (!BDS && BS && !IS && !EWR_15 && !ODS)
                return (0x22a);
            if (!BDS && BS && !IS && EWR_15 && !ODS)
                return (0x22f);
            if (BDS && !BS && !IS && !EWR_15 && !ODS)
                return (0x234);
            if (BDS && !BS && !IS && EWR_15 && !ODS)
                return (0x23a);
            if (!BDS && !BS && !IS && !EWR_15 && !ODS)
                return (0x1fe);
            if (!BDS && !BS && !IS && EWR_15 && !ODS)
                return (0x209);
        }
    }
    if ((mode == 7) && (reg == 4))
        return (0x21d);

    return(0x06);
}
```

```
/*****************************************************************************/
/* IR_PROM:                                                                  */
/*      Returns the starting address of the microprogram corresponding       */
/* to the target level instruction in the IR.                                */
/*****************************************************************************/

IR_PROM()
{
     int op_code;                    /* IR bits 15-12 */
     int mode_1;                     /* IR bits 5-3 */
     int mode_2;                     /* IR bits 8-6 */
     int reg_1;                      /* IR bits 2-0 */
     int reg_2;                      /* IR bits 11-9 */
     int size;                       /* IR bits 7-6 */

     int IR_0;                       /* IR bit 0 */
     int IR_1;                       /* IR bit 1 */
     int IR_2;                       /* IR bit 2 */
     int IR_3;                       /* IR bit 3 */
     int IR_4;                       /* IR bit 4 */
     int IR_5;                       /* IR bit 5 */
     int IR_6;                       /* IR bit 6 */
     int IR_7;                       /* IR bit 7 */
     int IR_8;                       /* IR bit 8 */
     int IR_9;                       /* IR bit 9 */
     int IR_10;                      /* IR bit 10 */
     int IR_11;                      /* IR bit 11 */
     int IR_12;                      /* IR bit 12 */
     int IR_13;                      /* IR bit 13 */
     int IR_14;                      /* IR bit 14 */
     int IR_15;                      /* IR bit 15 */

     IR_0 = state.PIR & 0x01;
     IR_1 = (state.PIR >> 0x01) & 0x01;
     IR_2 = (state.PIR >> 0x02) & 0x01;
     IR_3 = (state.PIR >> 0x03) & 0x01;
     IR_4 = (state.PIR >> 0x04) & 0x01;
     IR_5 = (state.PIR >> 0x05) & 0x01;
     IR_6 = (state.PIR >> 0x06) & 0x01;
     IR_7 = (state.PIR >> 0x07) & 0x01;
     IR_8 = (state.PIR >> 0x08) & 0x01;
     IR_9 = (state.PIR >> 0x09) & 0x01;
     IR_10 = (state.PIR >> 0x0a) & 0x01;
     IR_11 = (state.PIR >> 0x0b) & 0x01;
     IR_12 = (state.PIR >> 0x0c) & 0x01;
     IR_13 = (state.PIR >> 0x0d) & 0x01;
     IR_14 = (state.PIR >> 0x0e) & 0x01;
     IR_15 = (state.PIR >> 0x0f) & 0x01;


     mode_1 = (state.PIR >> 3) & 0x07;
     mode_2 = (state.PIR >> 6) & 0x07;
     reg_1 = state.PIR & 0x07;
     reg_2 = (state.PIR >> 9) & 0x07;
     size = (state.PIR >> 6) & 0x03;
     op_code = (state.PIR >> 12) & 0x0f;
```

```
switch (op_code) {
     case 0:
          /* ADDI */
          if (!IR_11 && IR_10 && IR_9 && !IR_8 && (mode_1 == 0))
               switch (size) {
                    case 0:
                         return (0x261);
                         break;
                    case 1:
                         return (0x265);
                         break;
                    case 2:
                         return (0x269);
                         break;
               }
          if (!IR_11 && IR_10 && IR_9 && !IR_8 && (mode_1 > 0))
               switch (size) {
                    case 0:
                         return (0x26d);
                         break;
                    case 1:
                         return (0x275);
                         break;
                    case 2:
                         return (0x27d);
                         break;
               }

          /* ANDI to CCR */
          if (state.PIR == 0x023c)
               return (0x294);
          break;
     case 1:
          break;
     case 2:
          break;
     case 3:
          break;
     case 4:
          /* DIVS.L, DIVSL.L */
          if (IR_11 && IR_10 && !IR_9 && !IR_8 && !IR_7 && IR_6
              && (mode_1 == 0))
               return (0x472);
          if (IR_11 && IR_10 && !IR_9 && !IR_8 && !IR_7 && IR_6
              && (mode_1 > 1))
               return (0x459);
          break;
     case 5:
          if (!IR_8 && (mode_1 == 0))
               return (0x284);
          if (!IR_8 && (mode_1 == 1))
               switch (size) {
                    case 1:
                         return (0x288);
                         break;
                    case 2:
                         return (0x28c);
                         break;
               }
```

```
        if (!IR_8 && (mode_1 > 1))
            return (0x28e);
        break;
    case 6:
        break;
    case 7:
        break;
    case 8:
        /* DIVS.W Dn, Dn */
        if (IR_8 && IR_7 && IR_6 && (mode_1 == 0))
            return (0x433);
        /* DIVS.W <ea>, Dn */
        if (IR_8 && IR_7 && IR_6 && (mode_1 > 1))
            return (0x453);
        /* DIVU.W Dn, Dn */
        if (!IR_8 && IR_7 && IR_6 && (mode_1 == 0))
            return (0x4b6);
        /* DIVU.W <ea>, Dn */
        if (!IR_8 && IR_7 && IR_6 && (mode_1 > 1))
            return (0x4a0);
        break;
    case 9:
        break;
    case 10:
        break;
    case 11:
        break;
    case 12:
        break;
    case 13:
        /* ADD */
        if ((size < 3) && (mode_1 > 1) && IR_8)
            return (0x24b);
        if ((size < 3) && !IR_8)
            switch (mode_1) {
                case 0:
                    return (0x240);
                    break;
                case 1:
                    return (0x243);
                    break;
                default:
                    return (0x247);
                    break;
            }

        /* ADDA */
        if ((size == 3) && (mode_1 == 0))
            return (0x251);
        if ((size == 3) && (mode_1 == 1))
            return (0x255);
        if ((size == 3) && !IR_8 && (mode_1 > 1))
            return (0x259);
        if ((size == 3) && IR_8 && (mode_1 > 1))
            return (0x25d);
```

```
                /* ADDX */
                if (IR_8 && (mode_1 == 0) && (size < 3))
                     return (0x2a5);
                if (IR_8 && (mode_1 == 1) && (size < 3))
                     return (0x2a8);
                break;
           case 14:
                /* ASR */
                if (!IR_8 && !IR_4 && !IR_3 && IR_5)
                     return (0x297);
                if (!IR_8 && !IR_4 && !IR_3 && !IR_5)
                     return (0x29b);
                if (!IR_11 && !IR_10 && !IR_9 && !IR_8 && IR_7 && IR_6)
                     return (0x29f);
                break;
           case 15:
                break;
      }
      return(0x006);
}
```

Appendix C

Microprogram Data

Appendix C
Microprogram Data

ADDRESS	ROUTINE	BEST	WORST
000	IFETCH_1	3	3
003	IFETCH_2	3	3
006	STOP	1	1
007	TRAP	1	1
010	Dn	1	1
012	An	1	1
014	(An)	1	1
016	(An)+	2	2
019	-(An)	2	2
01C	(d16,An)	2	2
01F	(d8,An,Dn.SIZE*SCALE)	3	3
023	(d8,An,An.SIZE*SCALE)	4	4
028	()	2	2
02B	(bd)	3	3
030	(Dn.SIZE*SCALE)	2	2
033	(An.SIZE*SCALE)	3	3
037	(bd,An)	3	3
03C	(bd,Dn.SIZE*SCALE)	4	4
042	(bd,An.SIZE*SCALE)	5	5
049	(An,Dn.SIZE*SCALE)	2	2
04C	(An,An.SIZE*SCALE)	2	2
04F	(bd,An,Dn.SIZE*SCALE)	4	4
055	(bd,An,An.SIZE*SCALE)	5	5
05C	([])	3	3
060	([bd])	5	5
067	([An])	3	3
06B	(od)	3	3
070	([bd,An])	5	5
077	([bd],Dn.SIZE*SCALE)	5	5
07E	([bd],An.SIZE*SCALE)	6	6
086	([bd],od)	7	7
090	([An],Dn.SIZE*SCALE)	4	4
095	([An],An.SIZE*SCALE)	5	5
09B	([An],od)	5	5
0A2	(Dn.SIZE*SCALE,od)	4	4
0A8	(An.SIZE*SCALE,od)	5	5
0AF	([bd,An],Dn.SIZE*SCALE)	5	5
0B6	([bd,An],An.SIZE*SCALE)	6	6
0BE	([bd,An],od)	7	7
220	([bd],Dn.SIZE*SCALE,od)	8	8
225	([bd],An.SIZE*SCALE,od)	9	9
0C8	([An],Dn.SIZE*SCALE,od)	6	6
0D0	([An],An.SIZE*SCALE,od)	7	7
0D9	([bd,An],Dn.SIZE*SCALE,od)	8	8
0E4	([bd,An],An.SIZE*SCALE,od)	9	9
0F0	([Dn.SIZE*SCALE])	4	4
0F5	([An.SIZE*SCALE])	4	4

0FA	([bd,Dn.SIZE*SCALE])		6	6
102	([bd,An.SIZE*SCALE])		7	7
10B	([An,Dn.SIZE*SCALE])		4	4
110	([An,An.SIZE*SCALE])		4	4
115	([Dn.SIZE*SCALE],od)		6	6
11D	([An.SIZE*SCALE],od)		7	7
126	([bd,An,Dn.SIZE*SCALE])		6	6
12E	([bd,An,An.SIZE*SCALE])		7	7
22A	([bd,Dn.SIZE*SCALE],od)		8	8
22F	([bd,An.SIZE*SCALE],od)		9	9
137	([An,Dn.SIZE*SCALE],od)		6	6
13F	([An,An.SIZE*SCALE],od)		6	6
148	([bd,An,Dn.SIZE*SCALE],od)		8	8
153	([bd,An,An.SIZE*SCALE],od)		9	9
15F	(d16,PC)		2	2
162	(d8,PC,Dn.SIZE*SCALE)		3	3
166	(d8,PC,An.SIZE*SCALE)		4	4
16B	(PC)		1	1
16D	(bd,PC)		3	3
172	(PC,Dn.SIZE*SCALE)		2	2
175	(PC,An.SIZE*SCALE)		2	2
178	(bd,PC,Dn.SIZE*SCALE)		4	4
17E	(bd,PC,An.SIZE*SCALE)		5	5
185	([PC])		3	3
189	([bd,PC])		5	5
190	([PC],Dn.SIZE*SCALE)		4	4
195	([PC],An.SIZE*SCALE)		5	5
19B	([PC],od)		5	5
1A2	([bd,PC],Dn.SIZE*SCALE)		5	5
1A9	([bd,PC],An.SIZE*SCALE)		6	6
1B1	([bd,PC],od)		7	7
1BB	([PC],Dn.SIZE*SCALE,od)		6	6
1C3	([PC],An.SIZE*SCALE,od)		7	7
1CC	([bd,PC],Dn.SIZE*SCALE,od)		8	8
1D7	([bd,PC],An.SIZE*SCALE,od)		9	9
1E3	([PC,Dn.SIZE*SCALE])		4	4
1E8	([PC,An.SIZE*SCALE])		4	4
1ED	([bd,PC,Dn.SIZE*SCALE])		6	6
1F5	([bd,PC,An.SIZE*SCALE])		7	7
234	([PC,Dn.SIZE*SCALE],od)		6	6
23A	([PC,An.SIZE*SCALE],od)		6	6
1FE	([bd,PC,Dn.SIZE*SCALE],od)		8	8
209	([bd,PC,An.SIZE*SCALE],od)		9	9
215	(XXX).W		2	2
219	(XXX).L		3	3
21D	#XXX		2	2
---	ABCD	Dn,Dn	---	---
---	ABCD	-(An),-(An)	---	---
240	ADD	Dn,Dn	3	3
243	ADD	An,Dn	4	4
247	ADD	<ea>,Dn	4	4
24B	ADD	Dn,<ea>	5	5
251	ADDA	Dn,An	5	5

255	ADDA	An,An		5	5
259	ADDA.W	<ea>,An		5	5
25D	ADDA.L	<ea>,An		5	5
261	ADDI.B	#<data>,Dn		3	3
265	ADDI.W	#<data>,Dn		3	3
269	ADDI.L	#<data>,Dn		3	3
26D	ADDI.B	#<data>,<ea>		7	7
275	ADDI.W	#<data>,<ea>		7	7
27D	ADDI.L	#<data>,<ea>		7	7
284	ADDQ	#<data>,Dn		3	3
288	ADDQ.W	#<data>,An		6	6
28C	ADDQ.L	#<data>,An		4	4
28E	ADDQ	#<data>,<ea>		6	6
2A5	ADDX	Dn,Dn		3	3
2A8	ADDX	-(An),-(An)		7	7
---	AND	Dn,Dn		3	3
---	AND	An,Dn		4	4
---	AND	<ea>,Dn		4	4
---	AND	Dn,<ea>		5	5
---	ANDI.B	#<data>,Dn		3	3
---	ANDI.W	#<data>,Dn		3	3
---	ANDI.L	#<data>,Dn		3	3
---	ANDI.B	#<data>,<ea>		7	7
---	ANDI.W	#<data>,<ea>		7	7
---	ANDI.L	#<data>,<ea>		7	7
294	ANDI	#<data>,CCR		5	5
---	ANDI	#<data>,SR		---	---
297	ASL,ASR	Dn,Dn		6	6
29B	ASL,ASR	#<data>,Dn		6	6
29F	ASL,ASR	<ea>		7	7
2B0	Bcc.B	<label>	(not taken)	2	2
2B0	Bcc.B	<label>	(taken)	4	4
2B3	Bcc.W	<label>	(not taken)	4	4
2B3	Bcc.W	<label>	(taken)	4	4
2B6	Bcc.L	<label>	(not taken)	4	4
2B6	Bcc.L	<label>	(taken)	4	4
2B9	BCHG	Dn,Dn		6	7
2C1	BCHG	Dn,<ea>		8	9
---	BCHG	#<data>,Dn		6	7
---	BCHG	#<data>,<ea>		9	10
---	BCLR	Dn,Dn		6	7
---	BCLR	Dn,<ea>		8	9
---	BCLR	#<data>,Dn		6	7
---	BCLR	#<data>,<ea>		9	10
---	BFCHG	Dn {offset:width}		---	---
---	BFCHG	<ea> {offset:width}		---	---
---	BFCLR	Dn {offset:width}		---	---
---	BFCLR	<ea> {offset:width}		---	---
---	BFEXTS	Dn {offset:width},Dn		---	---
---	BFEXTS	<ea> {offset:width},Dn		---	---
---	BFEXTU	Dn {offset:width},Dn		---	---
---	BFEXTU	<ea> {offset:width},Dn		---	---
---	BFFFO	Dn {offset:width},Dn		---	---
---	BFFFO	<ea> {offset:width},Dn		---	---

---	BFINS	Dn,Dn {offset:width}	---	---
---	BFINS	Dn,<ea> {offset:width}	---	---
---	BFSET	Dn {offset:width}	---	---
---	BFSET	<ea> {offset:width}	---	---
---	BFTST	Dn {offset:width}	---	---
---	BFTST	<ea> {offset:width}	---	---
---	BKPT	#<data>	---	---
2DB	BRA.B	<label>	3	3
2DD	BRA.W	<label>	3	3
2DF	BRA.L	<label>	4	4
---	BSET	Dn,Dn	6	7
---	BSET	Dn,<ea>	8	9
---	BSET	#<data>,Dn	6	7
---	BSET	#<data>,<ea>	9	10
2E2	BSR.B	<label>	7	7
2E7	BSR.W	<label>	7	7
2EC	BSR.L	<label>	8	8
2CB	BTST	Dn,Dn	4	4
2D3	BTST	Dn,<ea>	5	5
---	BTST	#<data>,Dn	4	4
---	BTST	#<data>,<ea>	6	6
---	CALLM	#<data>,<ea>	---	---
---	CAS	Dc,Du,<ea>	---	---
---	CAS2	Dc1:Dc2,Du1:Du2,(Dn1):(Dn2)	---	---
---	CAS2	Dc1:Dc2,Du1:Du2,(Dn1):(An2)	---	---
---	CAS2	Dc1:Dc2,Du1:Du2,(An1):(Dn2)	---	---
---	CAS2	Dc1:Dc2,Du1:Du2,(An1):(An2)	---	---
2F1	CHK.W	Dn,Dn	4	4
---	CHK.L	Dn,Dn	4	4
2F6	CHK.W	<ea>,Dn	6	6
---	CHK.L	<ea>,Dn	6	6
392	CHK2	<ea>,Dn	9	12
392	CHK2	<ea>,An	10	13
302	CLR	Dn	2	2
305	CLR	<ea>	5	5
309	CMP	Dn,Dn	2	2
315	CMP	An,Dn	3	3
319	CMP	<ea>,Dn	4	4
31D	CMPA.W	Dn,An	3	3
---	CMPA.L	Dn,An	3	3
321	CMPA.W	An,An	2	2
---	CMPA.L	An,An	2	2
324	CMPA.W	<ea>,An	4	4
---	CMPA.L	<ea>,An	4	4
329	CMPI.B	#<data>,Dn	3	3
---	CMPI.W	#<data>,Dn	3	3
---	CMPI.L	#<data>,Dn	3	3
32D	CMPI.B	#<data>,<ea>	5	5
---	CMPI.W	#<data>,<ea>	5	5
---	CMPI.L	#<data>,<ea>	5	5
332	CMPM	(An)+,(An)+	5	5
---	CMP2	<ea>,Dn	9	12
---	CMP2	<ea>,An	10	13
---	cpBcc	<label>	---	---

---	cpDBcc	Dn,<label>		---	---
---	cpGEN	<parameters>		---	---
---	cpRESTORE	<ea>		---	---
---	cpSAVE	<ea>		---	---
---	cpScc	Dn		---	---
---	cpScc	<ea>		---	---
---	cpTRAPcc			---	---
---	cpTRAPcc	#<data>		---	---
338	DBcc	Dn,<label>	(count > -1)	3	3
338	DBcc	Dn,<label>	(count = -1)	6	6
338	DBcc	Dn,<label>	(cc = true)	3	3
433	DIVS.W	Dn,Dn		50	52
453	DIVS.W	<ea>,Dn		51	53
472	DIVS.L	Dn,Dn		78	80
459	DIVS.L	<ea>,Dq		79	81
472	DIVS.L	Dn,Dr:Dq		78	83
459	DIVS.L	<ea>,Dr:Dq		80	85
472	DIVSL.L	Dn,Dr:Dq		78	80
459	DIVSL.L	<ea>,Dr:Dq		79	81
4B6	DIVU.W	Dn,Dn		48	48
4A0	DIVU.W	<ea>,Dn		48	48
472	DIVU.L	Dn,Dq		73	73
459	DIVU.L	<ea>,Dq		75	75
472	DIVU.L	Dn,Dr:Dq		75	75
459	DIVU.L	<ea>,Dr:Dq		77	77
472	DIVUL.L	Dn,Dr:Dq		73	73
459	DIVUL.L	<ea>,Dr:Dq		75	75
---	EOR	Dn,Dn		3	3
---	EOR	Dn,<ea>		5	5
---	EORI.B	#<data>,Dn		3	3
---	EORI.W	#<data>,Dn		3	3
---	EORI.L	#<data>,Dn		3	3
---	EORI.B	#<data>,<ea>		7	7
---	EORI.W	#<data>,<ea>		7	7
---	EORI.L	#<data>,<ea>		7	7
---	EORI	#<data>,CCR		5	5
---	EORI	#<data>,SR		---	---
33D	EXG	Dn,Dn		3	3
341	EXG	An,An		4	4
346	EXG	Dn,An		5	5
34C	EXT.W	Dn		3	3
---	EXT.L	Dn		3	3
---	EXTB.L	Dn		3	3
---	ILLEGAL			---	---
34F	JMP	<ea>		5	5
353	JSR	<ea>		8	8
35A	LEA	<ea>,An		5	5
---	LINK	An,#<displacement>		---	---
---	LSL,LSR	Dn,Dn		5	5
---	LSL,LSR	#<data>,Dn		5	5
---	LSL,LSR	<ea>		6	6
35D	MOVE	Dn,Dn		2	2
360	MOVE	An,Dn		3	3
364	MOVE	<ea>,Dn		4	4

368	MOVE	Dn,<ea>	6	6
---	MOVE	An,<ea>	6	6
36D	MOVE	<ea>,<ea>	8	8
374	MOVEA	Dn,An	3	3
378	MOVEA	An,An	3	3
37C	MOVEA	<ea>,An	4	4
381	MOVE	CCR,Dn	2	2
384	MOVE	CCR,<ea>	5	5
38B	MOVE	Dn,CCR	3	3
38E	MOVE	<ea>,CCR	4	4
---	MOVE	SR,Dn	---	---
---	MOVE	SR,<ea>	---	---
---	MOVE	Dn,SR	---	---
---	MOVE	<ea>,SR	---	---
---	MOVE	USP,An	---	---
---	MOVE	An,USP	---	---
---	MOVEC	Rc,Dn	---	---
---	MOVEC	Rc,An	---	---
---	MOVEC	Dn,Rc	---	---
---	MOVEC	An,Rc	---	---
399	MOVEM.W	reg. list,<ea> (control)	23 + 2n	
---	MOVEM.L	reg. list,<ea> (control)	23 + 2n	
---	MOVEM.W	reg. list,<ea> -(An)	23 + 2n	
---	MOVEM.L	reg. list,<ea> -(An)	23 + 2n	
---	MOVEM.W	<ea>, reg. list (control)	23 + 2n	
---	MOVEM.L	<ea>, reg. list (control)	23 + 2n	
---	MOVEM.W	<ea>, reg. list (An)+	23 + 2n	
---	MOVEM.L	<ea>, reg. list (An)+	23 + 2n	
---	MOVEP	Dn,(d,An)	---	---
---	MOVEP	(d,An),Dn	---	---
3CF	MOVEQ	#<data>,Dn	2	2
---	MOVES	Dn,<ea>	---	---
---	MOVES	An,<ea>	---	---
---	MOVES	<ea>,Dn	---	---
---	MOVES	<ea>,An	---	---
406	MULS.W	Dn,Dn	10	10
411	MULS.W	<ea>,Dn	10	10
415	MULS.L	Dn,Dl	11	12
427	MULS.L	<ea>,Dl	14	15
415	MULS.L	Dn,Dh:Dl	10	10
427	MULS.L	<ea>,Dh:Dl	13	13
406	MULU.W	Dn,Dn	10	10
411	MULU.W	<ea>,Dn	10	10
415	MULU.L	Dn,Dl	11	12
427	MULU.L	<ea>,Dl	14	15
415	MULU.L	Dn,Dh:Dl	10	10
427	MULU.L	<ea>,Dh:Dl	13	13
---	NBCD	Dn	---	---
---	NBCD	<ea>	---	---
3D2	NEG	Dn	2	2
3D5	NEG	<ea>	6	6
---	NEGX	Dn	2	2
---	NEGX	<ea>	6	6
3DB	NOP		2	2

---	NOT	Dn	2	2
---	NOT	<ea>	6	6
---	OR	Dn,Dn	3	3
---	OR	<ea>,Dn	4	4
---	OR	Dn,<ea>	5	5
---	ORI.B	#<data>,Dn	3	3
---	ORI.W	#<data>,Dn	3	3
---	ORI.L	#<data>,Dn	3	3
---	ORI.B	#<data>,<ea>	7	7
---	ORI.W	#<data>,<ea>	7	7
---	ORI.L	#<data>,<ea>	7	7
---	ORI	#<data>,CCR	5	5
---	ORI	#<data>,SR	---	---
---	PACK	-(An),-(An),#<adjustment>	---	---
---	PACK	Dn,Dn,#<adjustment>	---	---
3DE	PEA	<ea>	6	6
---	RESET		---	---
---	ROL,ROR	Dn,Dn	5	5
---	ROL,ROR	#<data>,Dn	5	5
---	ROL,ROR	<ea>	6	6
---	ROXL,ROXR	Dn,Dn	5	5
---	ROXL,ROXR	#<data>,Dn	5	5
---	ROXL,ROXR	<ea>	6	6
3E3	RTD	#<displacement>	5	5
---	RTE		---	---
---	RTM	Dn	---	---
---	RTM	An	---	---
3E7	RTR		6	6
3EC	RTS		5	5
---	SBCD	Dn,Dn	---	---
---	SBCD	-(An),-(An)	---	---
3F0	Scc	Dn	2	2
3F4	Scc	<ea>	6	6
---	STOP	#<data>	---	---
---	SUB	Dn,Dn	3	3
---	SUB	An,Dn	4	4
---	SUB	<ea>,Dn	4	4
---	SUB	Dn,<ea>	5	5
---	SUBA	Dn,An	5	5
---	SUBA	An,An	5	5
---	SUBA.W	<ea>,An	5	5
---	SUBA.L	<ea>,An	5	5
---	SUBI.B	#<data>,Dn	3	3
---	SUBI.W	#<data>,Dn	3	3
---	SUBI.L	#<data>,Dn	3	3
---	SUBI.B	#<data>,<ea>	7	7
---	SUBI.W	#<data>,<ea>	7	7
---	SUBI.L	#<data>,<ea>	7	7
---	SUBQ	#<data>,Dn	3	3
---	SUBQ.W	#<data>,An	6	6
---	SUBQ.L	#<data>,An	4	4
---	SUBQ	#<data>,<ea>	6	6
---	SUBX	Dn,Dn	3	3
---	SUBX	-(An),-(An)	7	7

3FA	SWAP	Dn	4	4
---	TAS	Dn	---	---
---	TAS	<ea>	---	---
---	TRAP	#<vector>	---	---
---	TRAPcc		---	---
---	TRAPcc.W	#<data>	---	---
---	TRAPcc.L	#<data>	---	---
---	TRAPV		---	---
3FF	TST	Dn	2	2
402	TST	<ea>	4	4
---	UNLK	An	---	---
---	UNPACK	-(An),-(An),#<adjustment>	---	---
---	UNPK	Dn,Dn,#<adjustment>	---	---